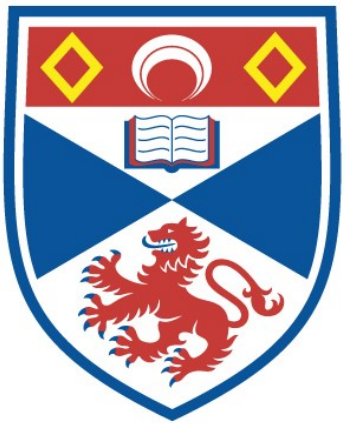
SYMMETRY IN CONSTRAINT PROGRAMMING

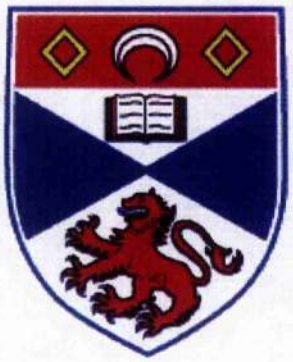
Iain McDonald

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2004

Symmetry in Constraint Programming



A thesis to be submitted to the
UNIVERSITY OF ST ANDREWS
for the degree of
DOCTOR OF PHILOSOPHY

by

Iain McDonald

School of Computer Science
University of St Andrews

September 2004

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

Constraint programming is an invaluable tool for solving many of the complex NP-complete problems that we need solutions to. These problems can be easily described as Constraint Satisfaction Problems (CSPs) and then passed to constraint solvers: complex pieces of software written to solve general CSPs efficiently.

Many of the problems we need solutions to are real world problems: planning (e.g. vehicle routing), scheduling (e.g. job shop schedules) and timetabling problems (e.g. staff rotas) to name but a few. In the real world, we place structure on objects to make them easier to deal with. This manifests itself as symmetry. The symmetry in these real world problems make them easier to deal with for humans. However, they lead to a great deal of redundancy when using computational methods of problem solving. Thus, this thesis examines some of the many aspects of utilising the symmetry of CSPs to reduce the amount of computation needed by constraint solvers.

In this thesis we look at the ease of use of previous symmetry breaking methods. We introduce a new and novel method of describing the symmetries of CSPs. We look at previous methods of symmetry breaking and show how we can drastically reduce their computation while still breaking all symmetry.

We give the first detailed investigation into the behaviour of breaking only subsets of all symmetry. We look at how this affects the performance of constraint solvers before discovering the properties of a good symmetry. We then present an original method for choosing the best symmetries to use.

Finally, we look at areas of redundant computation in constraint solvers that no other research has examined. New ways of dealing with this redundancy are proposed with results of an example implementation which improves efficiency by several orders of magnitude.

I, Iain McDonald, hereby certify that this thesis, which is approximately 50,000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date 20/10/04 *signature of candidate* _____

I was admitted as a research student in September 2000 and as a candidate for the degree of Doctor of Philosophy in September 2001; the higher study for which this is a record was carried out in the University of St Andrews between 2000 and 2003.

date 20/10/04 *signature of candidate* _____

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 20/10/04 *signature of supervisor* _____

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker.

date 20/10/04 *signature of candidate* _____

I, Iain McDonald, hereby certify that this thesis, which is approximately 50,000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date _____ *signature of candidate* _____

I was admitted as a research student in September 2000 and as a candidate for the degree of Doctor of Philosophy in September 2001; the higher study for which this is a record was carried out in the University of St Andrews between 2000 and 2003.

date _____ *signature of candidate* _____

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date _____ *signature of supervisor* _____

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker.

date _____ *signature of candidate* _____

Acknowledgements

I would like to thank mostly Ian Gent, Tom Kelsey, Steve Linton and Amy Midgley. This document would not have been possible without their help.

I would like to thank all the members of the APES research group, notably Lyndon Drake, Ian Miguel, Patrick Prosser, Evgeny Selensky, Dan Sheridan, and Barbara Smith and also to Warwick Harvey and Meinolf Sellmann.

I would like to thank Martin Bateman, Barry Stormont, Andrew Rowley, Scott Walker, Graeme Bell, Stuart Norcross, David Letham and Grant McLay.

Finally I would like to thank my family members Morgan, Margaret, Elizabeth, Tony and Alexander.

Published Research

Excerpts of this thesis have appeared in other forms in published research. We now list the research carried out during my time as a PhD student:

1. "Unique Symmetry breaking in CSPs using Group Theory" [McD01] by Iain McDonald appears in *Symmetry in Constraints*, pages 75-78, 2001.

2. "Partial Symmetry Breaking" [MS02] by Iain McDonald and Barbara M. Smith appears in *Principles and Practice of Constraint Programming*, pages 431-445. Springer, 2002.

3. "NuSBDS: Symmetry breaking made easy" [McD03] by Iain McDonald appears in *Symmetry in Constraint Satisfaction Problems*, pages 153-160, 2003.

4. "Symmetry and Propagation: Revising an AC algorithm" [GM03] by Ian P. Gent and Iain McDonald appears in *Symmetry in Constraint Satisfaction Problems*, pages 66-74, 2003.

5. "Conditional Symmetry in the All-Interval Series Problem" [GMS03] by Ian P. Gent, Iain McDonald and Barbara M. Smith appears in *Symmetry in Constraint Satisfaction Problems*, pages 55-65, 2003.

Apart from [GMS03] which does not appear in thesis, I am the primary author of all the above publications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many of the real world problems that we need solutions to are NP-complete problems so efficient techniques of solving them do not exist. These problems are pervasive in society e.g. scheduling, planning, configuration, circuit design, hardware verification, vehicle routing and timetabling. They can all be represented as constraint satisfaction problems (CSPs) and solved using constraint programming.

Constraint programming has become a popular method of solving the complex tasks mentioned above due to the natural way constraint problems are described. Also beneficial is that the problem description is all that is necessary, since the problem is solved by the computer.

However, constraint programming, no different from other combinatorial symbolic AI techniques, places human defined labelling on the problem to be solved. If we have n vehicles to schedule, we can see that each vehicle is identical to the next. In order to solve the problem though, we must label the vehicles, and in effect make each one different.

It is the labelling needed by constraint satisfaction problems that causes individual objects and patterns with the problem to be lost. These patterns, or symmetries, exist all around us. They put structure into the many things we see every day, making them easier for us to comprehend and work with.

This structure is lost to constraint programming. Once the labelling has been created, the symmetry disappears. This leads to a great deal of inefficiency when solving complicated

and symmetric problems.

Over recent years, research into how to deal with symmetry in CSPs has become very popular. Initially, attempts at using symmetry were generally dismissed as too complicated, leading constraint programmers to use more ad-hoc methods in practice.

One of the strengths of constraint programming is the vast collection of algorithms, search routines, heuristics and specialised constraints that research has developed. Currently, we are at an interesting point in the timeline of constraints research where methods of using symmetry are entering mainstream constraint programming.

This has come about by both a greater understanding of symmetry in CSPs as well as the development of effective ways of dealing with symmetry. The realisation of the constraint programming community that the pure mathematics area of group theory deals with the classification and measurement of symmetry has greatly furthered the ways in which we deal with symmetry in CSPs.

In this thesis we examine some aspects of the complicated relationship between symmetry and CSP solving. More specifically, we introduce one of the first group theoretic based methods of using symmetry in CSPs, we examine how using partial amounts of symmetry affects the complexity of CSP solving. We look at ways to bring symmetry use to mainstream constraint programming and by doing so we bridge the gap that exists between group theorists and constraint programmers. Finally we look at furthering the scope of symmetry use to beyond the combinatorial search at the heart of constraint programming. These form the main contributions of the thesis, discussed further at the end of the chapter.

1.1 Constraint Satisfaction Problems

A constraint satisfaction problem consists of a finite set of variables (or unknowns), each variable has a domain which is a finite set of possible values. There are also a set of constraints, each one forbidding a combination of assignments of values to variables. To solve a CSP, we must assign (or choose) a value for each variable from its domain such that none of the constraints are violated.

Consider the following problem: find five different numbers from 1 to 10 that sum to the

number 40. In this case, the unknowns are the five numbers we have to find, thus, these are the 5 variables of the problem.

$$A, B, C, D, E$$

Each number ranges from 1 to 10 so this is the domain of each variable.

$$A, B, C, D, E \in \{1..10\}$$

Finally, this problem has the following constraints.

$$\begin{aligned} A + B + C + D + E &= 40 \\ A \neq B \neq C \neq D \neq E & \\ A \neq C \neq E \neq B \neq D & \\ D \neq A \neq E & \end{aligned}$$

Given the above information, we can use a constraint solving toolkit to solve the above problem. Notice however that the variables each have the same domain and the constraints acting on them are commutative. Thus each variable is said to be *symmetric* to all the others i.e. if we find a solution to the above problem, we can re-arrange the values of the variables however we like to yield another solution. Though we can see this fact, the constraint solver cannot. Therefore, we may think of the following solutions as the same,

$$\begin{aligned} 6 + 7 + 8 + 9 + 10 &= 40 \\ 7 + 6 + 8 + 9 + 10 &= 40 \end{aligned}$$

but the constraint solver would not.

Finding a solution

In general, the above type of problem (or indeed any CSP) is solved by use of a combination of inference and search.

Each variable is considered in an order dictated by some heuristic. The chosen variable is then instantiated to some value taken from its domain. At this point, the set of domains of the variables may be passed to some propagation algorithm which can infer inconsistent choices that are then removed from the relevant domain. Before moving on to the next variable, the current set of instantiations is considered against the set of constraints to see if any of them have been violated. If any constraint has been violated, we backtrack from our previous decision and remove that choice from the relevant domain.

There are many different heuristics from a simple static lexicographic order based on our labelling, to the more complicated dynamically ordered smallest domain first. The latter heuristic works well in practice though there are other problems that work well with their own specific heuristics.

There are many different general purpose propagation algorithms which yield different levels of inference, as well as filtering algorithms for specific constraints. The aim of these algorithms is to detect and remove domain elements than cannot participate in a solution. These will be discussed in greater detail later.

Though there are a few backjumping algorithms available to the constraint programmer, a simple backtracking search procedure is used by default by most constraint solvers.

Since constraint solving is used in general to solve NP-complete problems, the runtimes are exponential. In the worst case, a complete backtrack search of all possible complete instantiations will have a runtime of $\mathcal{O}(d^n)$ for a CSP with n variables with domains of size d.

Symmetries and CSPs

We will now look at how we would solve the problem of finding five different numbers that sum to 40, using the approach described above. In this case we will not use any propagation algorithms and we will use a statically ordered lexicographic heuristic and a

simple backtracking search procedure.

We initially set the first variable, A, to be the first element in its domain.

$$\begin{aligned} A &= 1 \\ B, C, D, E &\in \{1..10\} \end{aligned}$$

We now recursively instantiate the next variable.

$$\begin{aligned} A &= 1 \\ B &= 1 \\ C, D, E &\in \{1..10\} \end{aligned}$$

As you can see, this violates the constraint that each number must be different. We therefore backtrack from this decision and remove the relevant value from the domain of the variable B.

$$\begin{aligned} A &= 1 \\ B &\in \{2..10\} \\ C, D, E &\in \{1..10\} \end{aligned}$$

We continue to exhaustively try all the different instantiations of variables B, C, D and E. After doing so we will discover that no solution exists with $A = 1$.

$$\begin{aligned} A &\in \{2..10\} \\ B, C, D, E &\in \{1..10\} \end{aligned}$$

However, by taking the symmetry of the problem into account we realise that no solution exists with *any* of the variables equal to 1. Therefore we can remove 1 from the domain of all variables.

$$A, B, C, D, E \in \{2..10\}$$

This is not something that a constraint solver would do though. By continuing our search to find a solution, we come across many inconsistent sets of instantiations that lead to failure. For each of these failures, we could use the symmetries of the problem to avoid making similar inconsistent sets of instantiations. After much redundant search we come to this first solution. If we were using some propagation algorithm to remove inconsistent assignments, they would also be performing redundant work.

$$\begin{aligned} A &= 6 \\ B &= 7 \\ C &= 8 \\ D &= 9 \\ E &= 10 \end{aligned}$$

Note that this is the only solution to this problem that is unique with respect to symmetry. Any other solution can be found by relabelling this solution e.g.

$$\begin{aligned} A &= 7 \\ B &= 6 \\ C &= 8 \\ D &= 9 \\ E &= 10 \end{aligned}$$

simply has the values of A and B exchanged. If we were to continue to search for solutions, which is a common thing to do in constraint programming, we would perform much more redundant search and find a further $5! - 1$ (or 119) superfluous, *duplicate* solutions.

Failing to appreciate the symmetries of a problem results in a large increase of computation both by increasing the search tree and the amount of reasoning done by propagation algorithms. The study of symmetry in constraint programming is an extremely important one as utilising the symmetry in highly symmetric problems is the only way to make them solvable.

1.2 Group Theory

Though the study of symmetry in CSPs is a fairly recent one, the study of symmetry in general dates back to the Ancient Egyptians who were aware of the symmetry of geometrical objects used in tiling. The classification of group theory as we know it today was introduced at the latter part of the 18^{th} century and is due to mathematicians such as Cauchy [OR97], Cayley [Cay78], von Dyck [vD82] and Burnside [Bur97] to name just a few.

To look at group theory in an abstract way, if we wish to measure the number of *occurrences* of something, we can use *integers*. If we wish to measure the *amount* of something, we can use *rational* numbers. If we wish to measure the amount of *symmetry* of an object, we can use *group theory*. The pure mathematics area of group theory is essentially the measurement and application of symmetry.

As an example of symmetry, imagine a square with the corners labelled (Figure 1.1). This has geometrical symmetry. We can rotate the square by 90°, 180° and 270° and the shape will remain the same. We can also invert or flip the square around the x-axis, y-axis, around the line bisecting the square diagonally from top left to bottom right, and from top right to bottom left.

We see that by applying any of these symmetries we change the corner labels but the resulting shape is still a square (Figure 1.2). This is exactly like the example mentioned at the start of the chapter: two vehicles may be conceptually identical but since symbolic AI labels them, the symmetry is lost.

Rather than thinking of an actual square, we can describe the symmetries as permutations of the labels. For example, the symmetry "rotate by 90°" can be written as a permutation of the points from $\{1, 2, 3, 4\}$ to $\{3, 1, 4, 2\}$.

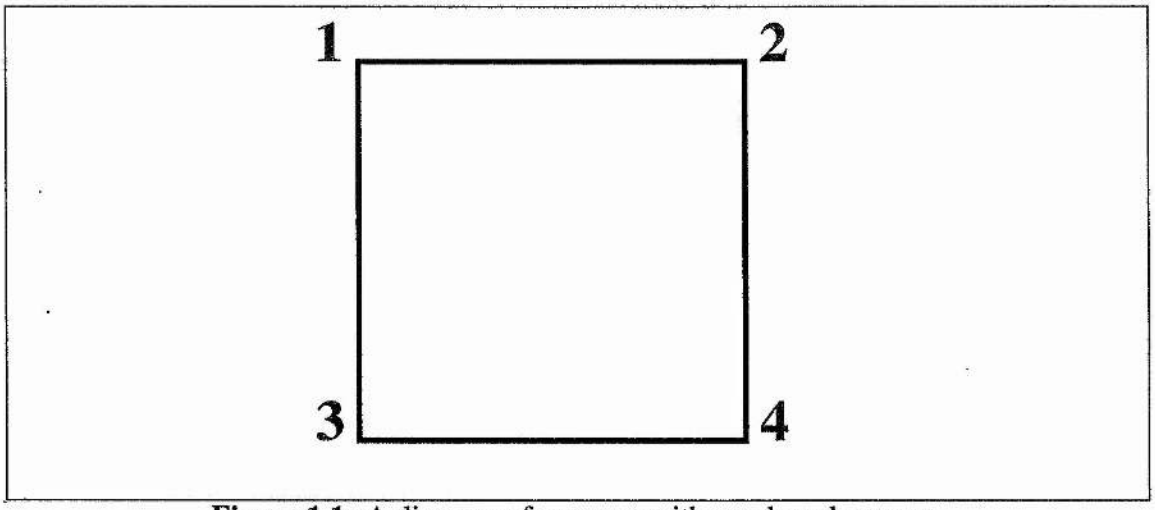


Figure 1.1: A diagram of a square with numbered corners.

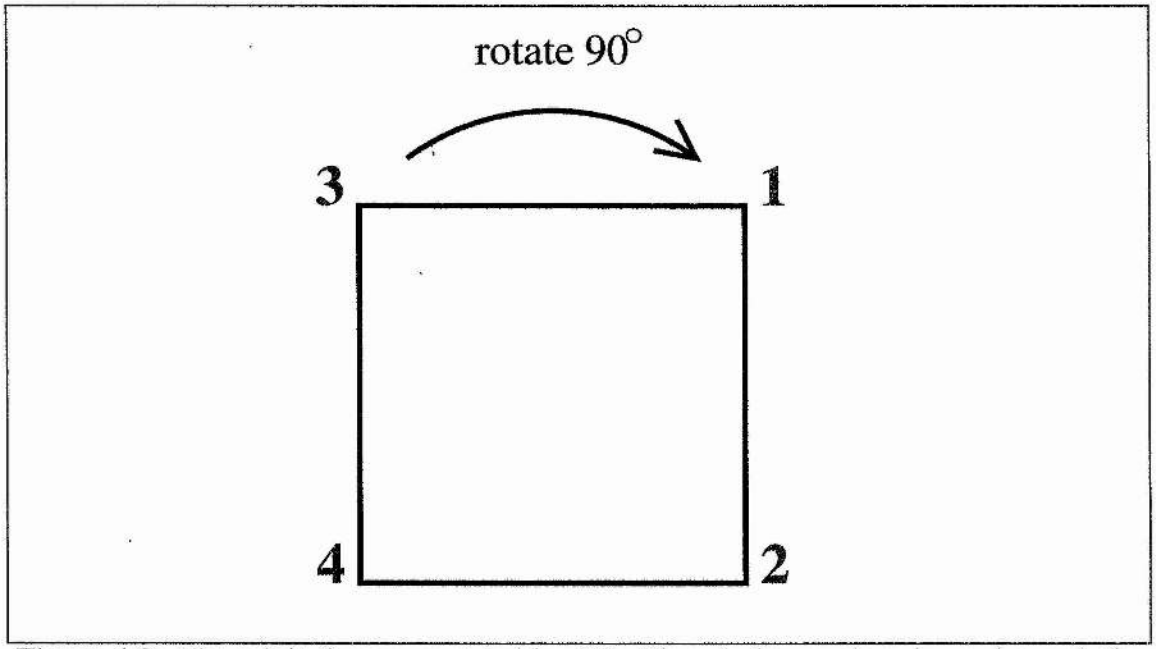


Figure 1.2: The original square rotated by 90°. Though the numbers have changed, the shape has not.

Rather than listing all the different symmetries explicitly, we can list a few symmetries and generate the remaining symmetries by combining the few we store [But91] [Ser03]. All the symmetries of a square can be generated by just the "rotate by 90°" and "invert about the x-axis" symmetries. For example, we can create the "invert about the y-axis" symmetry by combining the "rotate by 90°" symmetry twice and then applying the "invert about the x-axis" symmetry.

1.3 Definitions

We now define some of the preliminary ideas and concepts necessary before continuing with symmetry in constraint satisfaction problems.

Definition 1.1 *A CSP is a set of constraints C acting on a finite set of variables $\mathcal{X}$: $X_1..X_n$, each of which has a finite domain of possible values $D(X_i)$. A solution to a CSP L, is an instantiation of all the variables in $\mathcal{X}$ where $\forall i \; \exists j \; X_i = j, j \in D(X_i)$ such that all the constraints in C are satisfied.*

Definition 1.2 *A k-ary constraint is a constraint that acts on k variables of a CSP. A constraint on k variables $C_{1..k}$ describes the allowed combinations of choices allowed for the set of variables $\{X_1, ..., X_k\}$ as a subset of the cartesian product of $D(X_1) \times ... \times D(X_k)$.*

Definition 1.3 *A variable assignment (or instantiation) is a variable with a domain with only one element i.e. that variable has been assigned a value. A set of assignments is called a tuple (or partial assignment). An instantiation of all variables in a CSP is called a full assignment. For any given assignment or tuple, it is called consistent if it satisfies all the constraints acting of the variables in the assignment or tuple.*

Definition 1.4 *A nogood is an assignment or tuple that is inconsistent with at least one of the constraints of the CSP.*

Definition 1.5 *A solution to a CSP is a consistent instantiation of all the variables in the CSP. A partial solution is a tuple that does not violate any of the constraints acting on the variables in that tuple.*

The above are some of the fundamental terms we will use when dealing with CSPs. In general, when we are searching for a solution we are looking for the first solution, looking

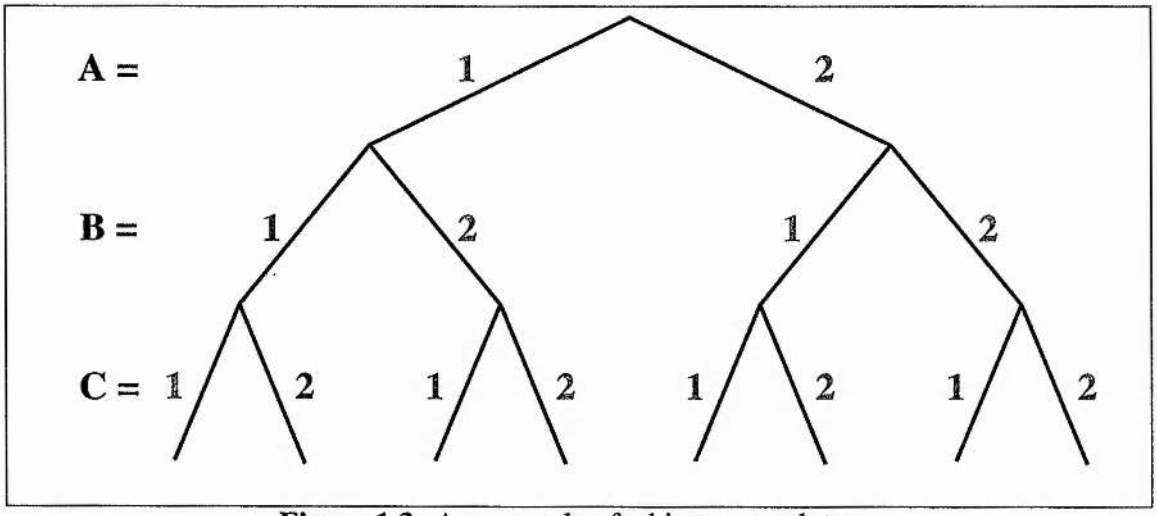


Figure 1.3: An example of a binary search tree.

for all solutions, verifying there are none, or looking for a solution that is optimised with respect to some criteria.

We find solutions to CSPs by traversing the *search space* of the CSP. This is all the possible combinations of tuples of the CSP. The path of the traversal of the search space is called the *search tree* (Figure 1.3).

While searching for a solution, we make decisions such as those described above. A positive decision is a variable assignment ($X_i = a$), and a negative decision is a domain removal ($X_i \neq b$).

Definition 1.6 *A state in search A, is the set of decisions made (positive and negative) while traversing the search space.*

1.3.1 Group Theory and Symmetry in CSPs

We now formally define a symmetry of a CSP. A symmetry is a bijective function. A function is bijective if it is both injective (no two inputs can have the same output), and surjective (every possible output can be reached by some possible input). The input and output of this function can be thought of as some representation of a CSP: a set of assignments, a single assignment, a set of domains etc.

Definition 1.7 *Given a CSP L, with a set of constraints C, a symmetry of L is a bijective*

function $f : A \rightarrow A$ where A is some representation of a state in search e.g. a list of assigned variables, a set of current domains etc., such that the following holds:

1. *Given A, a partial or full assignment of L, if A does not violate the constraints C, then neither does $f(A)$.*
2. *Similarly, if A is a nogood, then so too is $f(A)$.*

Recently, behaviour has been noticed in some problems which resembles symmetries but does not satisfy this definition. Conditional symmetries (also known as dynamic symmetries within planning research) are bijective functions that require a *condition* to be satisfied before they can be considered as symmetries [GMS03]. Pseudosymmetries are symmetries that are not bijective i.e. though f is a symmetry, its inverse f^{-1} may not be [PS98] [PB04].

Though Definition 1.7 describes a symmetry in terms of CSPs, we also want to think of a symmetry as an element of a group. Thus, we formally define a group.

Definition 1.8 *A group G is a non-empty set of elements with a binary operator $\times$, that obeys the following 4 axioms.*

1. *Every group contains an* identity *element.*
   $\exists e \in G \; s.t \; \forall g \in G, \; e \times g = g \times e = g$

2. *Every element of a group has an* inverse *element.*
   $\forall g \in G, \; \exists g^{-1} \in G \; s.t. \; g \times g^{-1} = g^{-1} \times g = e$

3. *The binary operator of a group is* associative.
   $\forall i, j, k \in G, \; (i \times j) \times k = i \times (j \times k)$

4. *The group is* closed *under the binary operator.*
   $\forall i, j \in G, \; i \times j \in G$

The size of a group is the number of elements it contains. For group G, its size is $|G|$.

Definition 1.9 *A* subgroup H, *of group* G, *is a group that shares the same binary operator as* G *and contains a subset of the elements of* G. *Given groups* N *and* M, *we say* M *is a subgroup of* N *by writing* $M \subseteq N$.

Throughout this thesis we will think of groups as permutations groups.

Definition 1.10 *A group whose elements are bijective permutations from a set to itself are called permutation groups.*

We consider the set referred to in Definition 1.10 to be a contiguous set of points of the range $1..p$. For example, consider a permutation group acting on 4 points: $\{1, 2, 3, 4\}$. One possible permutation of these points is to permute 1 with 2, 2 with 3, and 3 with 1. This creates the set {3, 1, 2, 4}. A concise representation of permutations is called the *cycle form*. This form lists the disjoint cycles that represent how the points are affected. The cycle form of the above permutation is (1, 2, 3)(4). We read this as "Point 1 goes to point 2, 2 goes to 3, 3 goes to 1. Point 4 goes to point 4." We usually omit cycles of length one, leaving just (1, 2, 3).

Some common groups include the *symmetric group* whereby any bijective permutation of the points is allowed, and the *alternating group* whereby any bijective permutation that can be represented as an even number of transpositions is allowed. For groups acting on n points, the symmetric group has $n!$ elements and the alternating group has $\frac{n!}{2}$.

Definition 1.11 *A generator set* G_{gen} *of group elements, represents the group* G *if every element of* G *can be recreated by some combination of elements in* G_{gen}.

Though the theoretical maximum number of generators needed to recreate the entire group is $log_2|G|$, typically just 2 to 6 elements are needed in a generator set. This theorem is well known to the group theory community, however for completeness the proof is outlined here.

Consider a generator set $S_k = (g_1, g_2, ..., g_k)$ for the group G_{S_k}. The group generated by $S_{k-1} = (g_1, g_2, ..., g_{k-1})$ is a subgroup of G_{S_k} since $S_{k-1} \subseteq S_k$. It follows then that $G_{S_1} \subseteq G_{S_2} \subseteq ... \subseteq G_{S_k}$. Recall that the size of every subgroup is a factor of the original group. Therefore, $G_{S_{k-1}}$ is at least half the size of G_{S_k}. $|G_{S_1}| \geq 2^1, |G_{S_2}| \geq 2^2, ..., |G_{S_k}| \geq 2^k$ or $k \leq log_2(|G_{S_k}|)$ i.e. the maximum number of generators in a minimal generator set for group G, is log_2 of the size of G.

Permutation groups are of particular interest to constraint programmers since it gives a means of translating between the group theory terminology of group elements (or permutations) acting on sets of points, to symmetries acting on sets of assignments. Thus, rather than considering permutation groups acting on a set of points, we consider permutation groups acting on sets of variables, or assignments. Though we describe the group operator $\times$ for combining group elements, when applying a group element to a point we will use the notation of a function i.e. $g(1)$.

Definition 1.12 *The* orbit *O of a (set of) point(s) A under a group G is the set of all (sets of) points that A can be mapped to.* $\forall g \in G,\ g(A) \in O$.

Even though the size of a group may be large, the size of the orbit of a set of points can be much smaller. The orbit of any one point, for example, cannot be more than the number of points the group acts on.

Definition 1.13 *The* stabilizer *G_A of a (set of) point(s) A, under the group G is a subgroup of G such that:* $\forall g \in G_A,\ g(A) = A$.

In many cases we will think of the orbit of one point under a group G, or the stabilizing subgroup of one point under a group G. However, when we consider more than one point, it is important to make a distinction between a *set of points* or a *list of points*. This is because the orbit of a set of points (called the setwise orbit) is different from the orbit of a list of points (called the pointwise orbit).

For example, given the set of points $\{1, 2\}$ with the symmetric group acting on 3 points, the setwise orbit is $\{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$. The pointwise orbit is $\{[1, 2], [2, 3], [1, 3], [2, 1], [3, 2], [3, 1]\}$. The setwise orbit is always a subset of the pointwise orbit.

The reverse is true of setwise stabilizers and pointwise stabilizers. Consider the diagram in Figure 1.4 which contains a matrix with freely permutable rows and columns. If we wish to stabilize the highlighted element, we must forbid any group element that permutes any column with column 2, any row with row 2, and any combination of those permutations.

If we calculate the stabilizer in an incremental way, this is the same as performing the pointwise stabilizer: $Stabilizer(Stabilizer(G, A), B) = PointwiseStabilizer(G, [A, B])$. Thus the pointwise stabilizer of the highlighted elements in Figure 1.5 would not allow any permutations of row 2 or row 4, or column 2, or any combinations of these permutations.

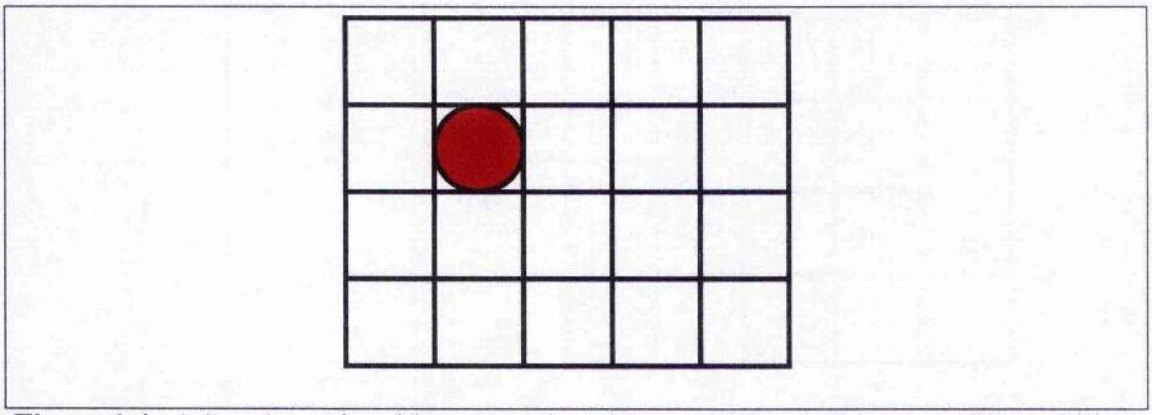


Figure 1.4: A 5×4 matrix with symmetric columns and symmetric rows. The stabilizer of the highlighted element forbids any permutations of column 2 and row 2.

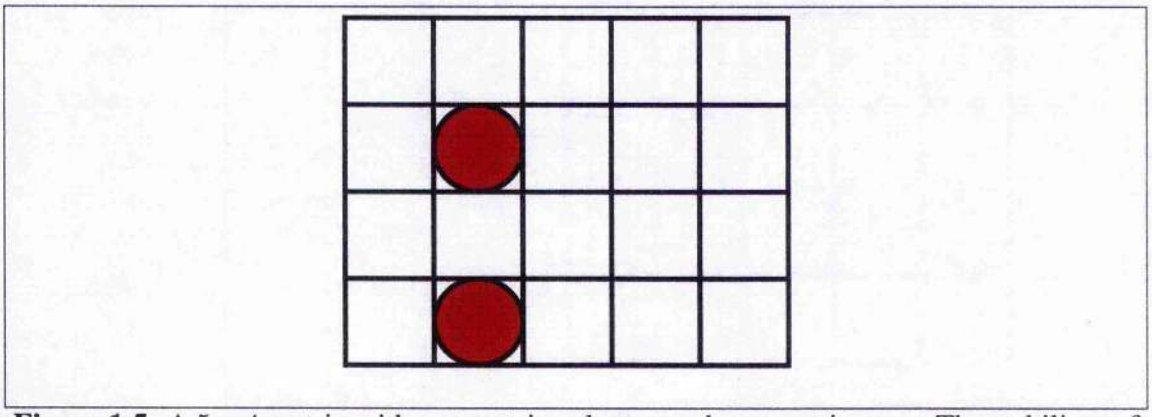


Figure 1.5: A 5×4 matrix with symmetric columns and symmetric rows. The stabilizer of the highlighted elements differs depending on whether it is setwise or pointwise stabilized.

However, the setwise stabilizer would allow the permutation of row 2 with row 4. Thus we can see that the pointwise stabilizer is a subgroup of the setwise stabilizer.

The complexity of pointwise group theory operations is less than setwise operations since they can be done incrementally. There is an algorithm [Ser03], however, that proposes recording the setwise orbit for a set of points A. The orbit of A can then be used to construct the orbit of B, where $A \subseteq B$, with less computation. This algorithm has yet to be implemented.

We will see more about the applications of permutation groups, orbits, stabilizers and how to calculate them later.

1.4 Contributions

Symmetries are prevalent in real world problems which are solved by constraint programming. As has already been shown, the lack of regard for these symmetries causes much redundancy when solving CSPs. For this reason, the use of symmetry while solving CSPs needs to be investigated thoroughly. Symmetry in CSPs is a vast research area with frequent new avenues of study being introduced. This thesis looks at the problems of symmetry in constraint programming from several angles and contributes to each of them.

From a constraint programmer's point of view, they are not concerned with the internal workings of their constraint solver. They simply wish to be able to describe their problems easily and have their constraint solver find solutions as efficiently as possible. Much of the research into CSPs culminates in implementations of effective constraint solvers. These solvers take the best algorithms from the many areas within CSP research and make them easy to use.

In this thesis, we look at the problem of symmetry in CSPs from the point of view of the constraint programmer. We discuss the problems with the most popular methods of breaking symmetry in terms of ease of use. We present two new implementations of symmetry breaking systems that advance the inclusion of symmetry breaking into constraint solvers. Firstly, we introduce U-SBDS which has the advantage of a concise symmetry representation. This leads to less effort in describing symmetries. U-SBDS also limits the number of symmetry breaking constraints needed to break all symmetry. Secondly, we introduce NuSBDS which contains an intuitive method of describing symmetries that has many advantages over all previous methods.

For highly symmetric problems that have an exponential amount of symmetry, there is currently no method to efficiently break all of it[1]. When dealing with such problems, only a subset of all possible symmetry can be dealt with. Though many symmetry breaking methods can be used to break subsets of symmetry, no detailed research into breaking such subsets has previously been carried out.

In this thesis we look at how the amount of symmetry broken affects the amount of work required in finding solutions. From this investigation we look at why some subsets of symmetry appear to be better than others. We then present an algorithm for choosing the

[1] Apart from value symmetry [RDGKL04].

best symmetries.

Almost all previous research into symmetry in CSPs has looked at how the symmetry affects search. This is done almost exclusively by adding constraints to the original CSP, or by adding constraints dynamically during search to the solver. We look at trying to use symmetry in CSPs to reduce the amount of work done in other areas of CSP solving. This is the first research to do this. Based on this study we present a modified propagation algorithm that results in fewer constraint checks by many orders of magnitude. We also suggest many ways in which other redundant work could be reduced.

To summarise, the main contributions of this thesis are:

1. An evaluation of current symmetry breaking systems and a discussion of their suitability for use within constraint solvers.

2. A new method of breaking symmetry while constraint solving using group theory, U-SBDS.

3. An analysis of the number of required symmetry breaking constraints for breaking symmetry dynamically during search. We introduce an upper-bound on the number of symmetry breaking constraints needed after any one backtrack.

4. A new and novel method of easily describing problem specific symmetries, NuSBDS.

5. The first comprehensive study of breaking subsets of symmetries in highly symmetric problems. We then discover the properties of a good symmetry and produce an algorithm for selecting the best symmetries to use when breaking symmetries during search.

6. A realisation of new ways in which symmetry can be used to reduce the computation of solving CSPs. We modify an existing constraint programming algorithm to take advantage of symmetry and present empirical results.

1.5 Thesis Outline

The rest of this thesis is organised into the following chapters. Chapter 2 will review the previous work in the field of constraint programming including modelling, constraints, propagation and search. We will then look at symmetry breaking in constraint programming, computational group theory and the continuing convergence of the two areas.

Chapter 3 looks at various symmetry breaking systems from the point of view of the constraint programmer. We examine the advantages and limitations of the previous implementations of these systems and argue what the most desired elements of the ideal symmetry breaking system are. We introduce and evaluate two new implementations of symmetry breaking systems.

Chapter 4 contains an investigation into partial symmetry breaking. We look at previous research and discuss how their methods of breaking symmetry were limited in order to minimise run-times. We then present experimental evidence of how breaking varying numbers of symmetry affects the overall run-time of solving symmetric problems. We conclude this chapter by presenting an algorithm for selecting the best symmetries to use when symmetry breaking along with supporting empirical evidence.

Chapter 5 argues that we should re-use all information gathered during search and not just failed search decisions as we currently do. We present a modified definition of symmetry which we use to take account of this. A refinement to a propagation algorithm is presented before implementing a modified arc consistency algorithm. Empirical results of using this modified algorithm are presented.

Finally, we conclude with Chapter 6. We reiterate the contributions of this thesis and discuss future directions of research.

Appendix B contains the user manual for the symmetry breaking system: NuSBDS (Chapter 3.5). Appendix C contains a reference for many of the common CSPs featured in this thesis. As well as suggested models, it more importantly contains details of the amount of symmetry of certain problems.

Chapter 2

Review of Previous Work

We now review previous work related to this thesis. This work will fall roughly into three categories. The first of these is related to the discipline of constraint programming, how it is used in practice as well as some issues of note for constraint programmers. Secondly, we will briefly look at *computational group theory* (CGT) which is concerned with implementing efficient group theoretic operations. Finally we will look at previous methods of symmetry breaking and other investigations into symmetry specifically in CSPs.

2.1 Constraint Programming

The ethos behind the declarative programming paradigm, in which constraint programming lies, is a unique one. Whereas other programming paradigms require the programmer to describe how to solve a problem which the computer then automates, constraint programming requires the programmer to describe the problem which the computer then solves. Though the types of problems that can be solved by constraint programming are generally limited to NP-complete combinatorial search problems, the ease with which they can be solved is very beneficial to programmers.

For this advantage, a new approach to solving problems is needed i.e. a new programming paradigm. Broadly speaking the methodology for solving CSPs is broken into three sections. The first is the *modelling* stage. Here we define the structure of what a solution looks like i.e. what types of variable we have and what their possible values are. Secondly, we

constrain the model. In this stage, we describe the constraints which forbid combinations of assignments i.e. the problem description itself. Finally, we must *search* for a solution. This stage is concerned with how we traverse the search space or more specifically which heuristics and search routines we use. We can also state what level of propagation we wish to enforce during this search and what algorithms we will use to accomplish this.

Each of these stages are interconnected in that changes to one will affect the others. We will now look in more detail at these three stages as well as other areas important to constraint programmers when solving CSPs.

2.1.1 Modelling

The modelling stage is one the most important in terms of tractability since how we decide to model our problem has far reaching effects on the efficiency of our program. As such, workshops and other research within the constraint programming community are devoted to modelling.

Modelling is sometimes referred to as the "bottleneck" in constraint programming, since sometimes expert knowledge and experience is needed to model problems effectively. It is this knowledge that researchers are trying to capture to make constraint programming a more easily accessible tool for industry and academics in general.

In its basic form, modelling is concerned with looking at a problem and identifying the unknowns and realising what their possible values could be i.e. describing the variables and their domains. The main work of this thesis is concerned with *finite integer variables* i.e. each variable has a domain which is a finite set of integers. Generally speaking this set of integers is contiguous from 0 or 1 to n, though it does not need to be.

As well as integer variables we can use set variables [ILO00], whose domain is a set of integers and whose value can be any subset of its domain. Particularly relevant to scheduling problems, we can also use interval variables [BPN01]. Here the domain of such variables is a range between two floating point numbers and its value is a floating point number that satisfies the constraints to within some predefined degree of error.

We may also wish to include so called *redundant models* whereby we use more than one model for the problem [CLW99]. In such cases the models are combined by *channelling*

constraints. These constraints are used to ensure that the two models yield the same answer. By including redundant models the search space is increased exponentially, however, the search tree is not affected since the channelling constraints ensure the models behave consistently. The main reason for using redundant models is that some constraints may be easier to express and/or more efficient to deal with when used with a specific model. A good example of this in practice is featured in [DdVC03].

For problems like the sum problem found in the previous chapter, the modelling stage is quite straightforward, we simply had a list of 5 finite domain integer variables. However more complicated problems have many different ways of being modelled.

Example 2.1 *n-queens problem. Given an $n \times n$ chessboard, we must place n queens on the board such that none of them can attack another piece i.e. no two queens appear in the same horizontal, vertical or diagonal line.*

The problem in Example 2.1 has many different ways of being modelled [Nad90]. The reason for this is that we can describe what is unknown in many ways. We can say that the square on the board that each queen takes is unknown. We can say that whether or not a queen goes on a square is unknown. We could also say that the column position of the queen in each row is unknown. Each of the descriptions above creates different models and more importantly, each model has a different search space. Also, different models may be easier to describe constraints for.

Model 2.1

The unknowns in this case (and therefore the variables) are the positions of the queens. The possible values that each queen can take (and therefore the domain of each variable) are the squares on the board. For the n-queens problem, this results in n variables each with a domain of n^2 squares. Thus the size of the search space is $(n^2)^n$. For $n = 8$ this is 2.8×10^{14}.

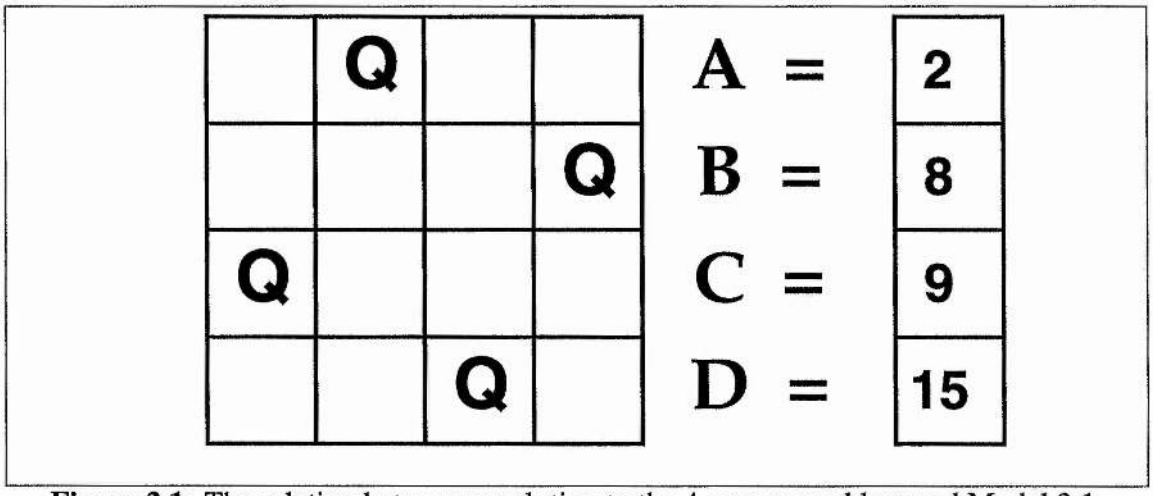


Figure 2.1: The relation between a solution to the 4-queens problem and Model 2.1.

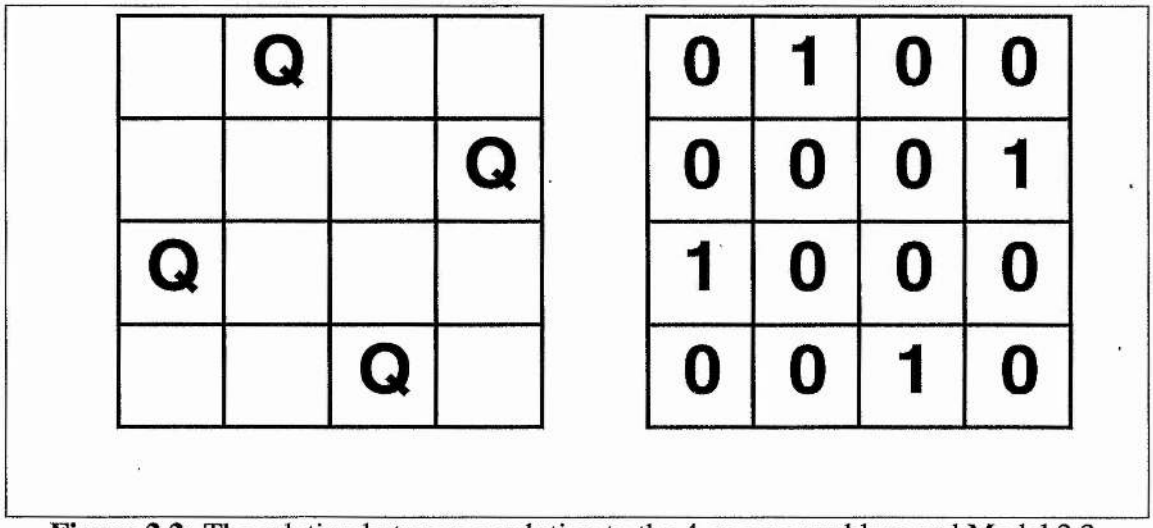


Figure 2.2: The relation between a solution to the 4-queens problem and Model 2.2.

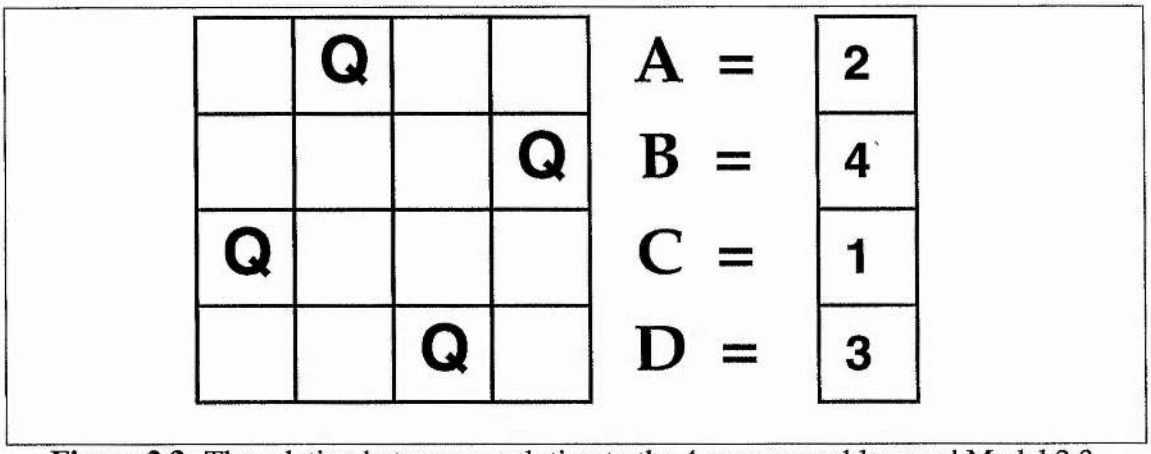


Figure 2.3: The relation between a solution to the 4-queens problem and Model 2.3.

Model 2.2

The unknowns in this case are the contents of each square. The possible values that each square can take is the existence or non-existence of a queen. For this model, there are n^2 variables each with a domain of two. Thus the size of the search space is 2^{n^2}. For $n = 8$ this is 1.8×10^{19}.

Model 2.3

By noticing that each row has to have exactly one queen on it, we can have a variable for each row and its value is the column the queen appears in. In this case we will have n variables, each with a domain of size n resulting in a search space of n^n. For $n = 8$ this is 1.7×10^7.

We can see, even with a very simple problem such as the n-queens, the modelling choices can result in large differences in the search space and therefore the time taken to solve the problem. Different models need to be constrained differently and we will see later how specialised constraints can also make a difference to the time taken to the solve the problem.

Modelling and Symmetry

With respect to symmetry, modelling is one of the most important aspects of constraint programming. The model we choose not only affects the search space and how easily the problem can be constrained but it also dictates the number of symmetries that the problem has.

Some problems inherently have symmetry that cannot be avoided, however, poor modelling can introduce needless symmetry. Similarly, a good model can reduce the amount of symmetry a problem has [Smi01].

We will now look at a simple example of this by examining the above n-queens models. Given a solution to Model 2.1, we can freely permute the variables to yield another solution. Also, the geometrical property of the square - on which the n-queens problem is based - has 8 symmetries. Thus, this model has $8n!$ symmetries. The interchangeable symmetry does not exist in Model 2.3, though the symmetries of the square still do. Thus, this model has 8 symmetries regardless of the size of n. So as well as Model 2.1 having a larger search space to Model 2.3, it also has exponentially more symmetries.

Matrix Models

Many problems that can be solved by constraint programming can be modelled as matrices [FFH+01]. A matrix model can be thought of as an $n \times m$ array of variables where the rows and columns are considered to be symmetric i.e. the rows can be permuted freely as can the columns. For models such as this, the resulting CSP will have $n! \times m!$ symmetries. Though this seems like a poor modelling decision, in many cases there are not any more preferable alternatives and we must deal with the symmetries in other ways.

As well as the other general methods mentioned in this thesis (such as those presented in Chapter 2.4) there are techniques that can be used specifically for breaking symmetries in matrix models. These techniques involve adding ordering constraints [FFH+02] for which there are GAC algorithms[1] [FHK+02] [CB02]. A constraint encoding was described in [GPS02] for matrix models which performs the same domain pruning as the global constraint in [FHK+02].

[1] This term will explained later

The main idea behind these ordering constraints is that by adding superfluous constraints to the model (which rule out some solutions, but not all) we are destroying the symmetry of the problem by stating that rows or columns of variables are different. There are different ways we can order such matrices, for instance using lex constraints, ordered multisets, or either of the previous methods with ordered sums. A detailed analysis of these methods is presented in [Kiz04].

These methods need an exponential number of constraints to break all the $n! \times m!$ symmetries of a matrix [CGLR96]. Since adding this number of constraints to the model is not computationally tractable, a subset of the symmetries is broken by adding a polynomial subset of constraints.

2.1.2 Constraints

Describing the constraints of a CSP is where most effort is exerted by the constraint programmer. It is the constraints that essentially describe the problem. As we saw in Definition 1.2, constraints act on a set of variables. We can think of a constraint as a list of allowed of tuples, a list of forbidden tuples, or some implicit function that takes a tuple and returns valid, invalid or possibly unknown if a partial assignment is being considered [BR97].

In constraint solvers, constraints are usually constructed using combinations of arithmetic and logical operators e.g. $=$, $\neq$, $\leq$, $+$, $\wedge$ etc. All of these are binary operators, which on their own, can create binary constraints. CSPs that contain just binary constraints are called binary CSPs. Any CSP containing a constraint with arity larger than 2 is called a non-binary constraint. It has been shown that any non-binary CSP can be transformed into a binary CSP [RPD90]. Generally speaking, propagation (Chapter 2.1.3) is easier to perform on constraints with a small arity. Thus, research has looked at converting non-binary CSPs to binary CSPs [Ste01] [BCvW02].

We have already observed how constraint programming allows its users to easily describe complex problems. One of the main tools of constraint programming that makes this so is *global constraints*. A global constraint is a constraint that acts over a set of variables, thus it is also a non-binary constraint. A global constraint also consists of a *filtering algorithm*, which is a propagation algorithm to be used exclusively just for the global constraint in question. Examples of global constraints are all-different: given n variables, ensure that

each one takes a different value. Occurrences: given n variables, a number of occurrences o and a domain element i, ensure that o of the n variables take the value i. Since the idea of global constraints is too closely tied to propagation, we will return to these constraints after we have looked at propagation in more detail.

2.1.3 Propagation

One of the many reasons that constraint programming is able to solve complex problems efficiently, is due to the careful balance of propagation and search. We can reduce the exponential search space by using propagation algorithms during search to prune branches of the search tree. The importance of propagation has led to much research into both different levels of consistency and different ways to achieve them. Note that the term propagation is sometimes also referred to as consistency, inference or filtering.

The main idea behind propagation algorithms is to take a current state in search and a set of constraints and remove values from the domains of variables we can show to be inconsistent. For example, imagine the constraint $C_{ij} : X_i + X_j = 5$ where $D(X_i) = D(X_j) = \{1, 2, 3\}$. We can see that if $X_i = 1$ then there is no value that X_j can take that will satisfy C_{ij}. Therefore, we conclude that $X_i = 1$ is an *inconsistent* choice and thus we remove 1 from $D(X_i)$. For the same reason we can infer that $X_j = 1$ is also inconsistent and thus 1 is also removed from $D(X_j)$. Constraint solvers are good at making simple inferences such as these. After making such a domain removal, we can then *propagate* these changes i.e. see if we can make any further inferences based on the new smaller domains. By performing these computationally cheap inferences, we drastically reduce the size of the search space.

A propagation algorithm enforces a *level of consistency*. A consistency level is not concerned with how the propagation algorithm is implemented, but what the result of applying it to a set of domains is. We will now define some levels of consistency and provide references to some implementations of algorithms that enforce them.

Any time we instantiate a variable X_i, we examine the constraints that act on that variable. To perform *forward checking* we find any instantiations $X_j = a$, that would result in failure if they were to be made in conjunction with the newly instantiated X_i. When we find such an instantiation as $X_j = a$, we remove a from $D(X_j)$. This is one of the simplest levels

of consistency to enforce. The most common form of consistency used with constraint programming is *arc consistency*. One reason for this is that it is the strongest level of propagation that can be enforced by adding unary constraints, or in other words, a domain removal.

Definition 2.1 *A binary constraint C_{ij} is arc consistent iff:*

1. $\forall a \in D(X_i)\ \exists b \in D(X_j)\ s.t.\ (a,b) \in C_{ij}$

2. $\forall b \in D(X_j)\ \exists a \in D(X_i)\ s.t.\ (a,b) \in C_{ij}$

Definition 2.2 *A binary CSP L is arc consistent iff all constraints in L are arc consistent.*

The study of arc consistency was introduced by Mackworth in [Mac77]. This paper contained an algorithm for enforcing arc consistency on binary CSPs called AC-3. This algorithm has time complexity $\mathcal{O}(ed^3)$ and space complexity $\mathcal{O}(e+nd)$ where e is the number of constraints and d is the size of the largest domain. Since then, other implementations with the optimal time complexity $\mathcal{O}(ed^2)$ and space complexity $\mathcal{O}(ed)$ (except for AC-4 which has space complexity $\mathcal{O}(ed^2)$) have been created: AC-4 [MH86], AC-6 [BC93], AC-7 [BFR95], AC-2001 [BR01], AC-3.1 [ZY01]. In practice, AC-7 performs the smallest number of constraint checks and is in many ways preferable. The algorithms AC-2001 and AC-3.1 are the same, though developed independently, and are similar to AC-3. Though the complexities of the above algorithms are similar, how well they work in practice has been an area of interest [Wal93] [vD03].

All of the above algorithms apply only to binary constraints though. Given that constraint solvers allow the construction of larger constraints, and also global constraints, we need to be able to enforce arc consistency on non-binary constraints. This level of consistency is call *generalised arc consistency* or GAC. An algorithm for enforcing GAC on a set of arbitrary constraints is presented in [BR97] and is loosely based on AC-7. This algorithm is called GAC-Schema and has time complexity $\mathcal{O}(ed^k)$ and space complexity $\mathcal{O}(ek^2d)$ and is thus quite impractical for constraints with a large arity.

Enforcing arc consistency means that we must verify that every possible domain element of every variable can be extended by at least one more decision. A limited version of this

consistency is also affective. A CSP is *bounds consistent* if we ensure that the largest and smallest element of every domain is consistent (Definition 2.3).

Definition 2.3 *A binary constraint C_{ij} is bounds consistent iff:*

1. $a \in \{D(X_i)_{smallest}, D(X_i)_{largest}\}\ \exists b \in D(X_j)\ s.t.\ (a,b) \in C_{ij}$

2. $b \in \{D(X_j)_{smallest}, D(X_j)_{largest}\}\ \exists a \in D(X_i)\ s.t.\ (a,b) \in C_{ij}$

For some constraints e.g. a binary constraint involving an addition operator, performing bounds consistency is as powerful as arc consistency for contiguous domains, but computationally cheaper.

A concise representation of some levels of consistency can be described by the following definition.

Definition 2.4 *A CSP is (i,j)-consistent if we can make any i consistent instantiations and this state can be extended to a consistent instantiation of another j variables.*

We can see that arc consistency can be described as $(1,1)$-consistency. In addition, $(2,1)$-consistency is also known as path consistency, $(1,2)$-consistency is also known as path inverse consistency and $(k-1,1)$-consistency is also known as k-consistency. In general, we need to perform searches over j variables, and post constraints of arity i, to enforce (i,j)-consistency.

Global Constraints

As mentioned earlier, a global constraint is one that can act on any number of variables, possibly even all the variables in a CSP. However, Chapter 2.1.3 showed that such constraints cannot be propagated efficiently in general. Global constraints are such an important part of constraint solver though, that research is conducted to find efficient methods of enforcing a level of consistency for specific constraints. These specific algorithms take advantage of the semantics of the individual constraints.

One of the most popular global constraints is the *all different* constraint, for which there is an algorithm that enforces GAC in polynomial time [Reg94]. Other examples of global constraints include the *global cardinality constraint* [Reg96] and *lexicographic ordering constraints* [FHK+02].

2.1.4 Heuristics

The next variable to instantiate while searching is decided by the heuristic we use. Though CSPs have an exponential search space, the size of the search tree can be reduced by propagation, and also by heuristics. Indeed, given the perfect heuristic for a given CSP, we can find a solution in polynomial time if there is one. In general this is not the case. However, good heuristics can greatly reduce the time taken to find a solution.

The traits of a good heuristic are that it guides the traversal of the search tree toward solutions, and that it can quickly find paths that lead to failure. Examples of dynamic heuristics include the *smallest domain first* which instantiates the variable with the smallest domain first, and the *most constrained first* which instantiates the variable with the most constraints acting on it first. Specific problems can have their own heuristics e.g. the n-queens problem (Model 2.3) can be effectively solved using a heuristic that tries to place queens nearest the center of the board, where it is most constrained. This is done by instantiating the variable nearest to $X_{\frac{n}{2}}$ with the value nearest to $\frac{d}{2}$ first [CHS+03]. A detailed empirical study of dynamic variable ordering heuristics can be found in [GMP+96] and an investigation into why certain variable heuristics are good can be found in [Smi97] [SG98].

As we will see later in this chapter, heuristics can play an important role in solving CSPs with symmetry breaking methods. It was shown in [GHK02] how a poor ordering heuristic can negate the benefit of some methods of symmetry breaking. One of the advantages of the symmetry breaking method described in [GS00], and later in other symmetry breaking methods such as that in [FSS01], is that they do not interact badly with the heuristic used.

This becomes particularly important when we are only interested in finding one solution. Generally, symmetry breaking methods are used to find all solutions or an optimal solution as this produces the greatest comparative reduction in search. The reverse is true of good heuristics. Though they try to explore the search space that will yield a solution first, the rest of the search space must still be explored later. Thus, the effect of a good heuristic

is lessened when finding all solutions or an optimal solution. Although, note that a good heuristic will try to search the remaining search in such a way that inconsistent subtrees are found sooner. Since symmetry breaking methods work best when finding all solutions, little symmetry breaking research has been carried out into finding just the first solution [Pre01].

To eliminate the variability of the effect that dynamic ordering heuristics may have, we will try to avoid their use in experiments. Therefore, unless stated otherwise, experiments in this thesis will use a static lexicographic ordering heuristic.

2.1.5 Search

There are many different strategies for traversing the search space of combinatorial search problems. CSPs are generally solved using a simple backtracking algorithm, and this is the case for the entirety of this thesis. Since it is an important aspect of constraint programming however, we will briefly discuss some alternatives.

A search strategy can be divided into one of two categories: complete and incomplete search. The former deals with methods that systematically traverse the entire search space of the problem until a solution is found. The latter may not traverse the entire search space. Complete search has the advantage of being guaranteed to find a solution, or prove that none exist. Incomplete search can either find a guaranteed solution or state that none were found, but may still exist.

Complete search records the domains of variables. Instantiations are made and domains are altered. States in search are verified to be consistent before recursively searching further, or we backtrack to the last consistent state to make a different instantiation. Backjumping methods work in a similar manner except they can backtrack further up the search tree [Gas79] [Dec90] [Pro93].

Incomplete search methods work by considering the leaves of the search tree i.e. complete instantiations. Many methods of incomplete search can be thought of as a general instance of local search. These include hill-climbing, tabu-search [Glo89], GSAT [SLM92] and stochastic search. For example, the main idea of GSAT is to try a random instantiation of all the variables (possibly guided by some heuristic). We then alter the instantiations

of individual variables (also called a *flip*) to try to minimise the number of violated constraints. If we cannot satisfy all of the constraints within a certain number of flips, we then (randomly) choose another complete instantiation of variables. There may also be a limit on the number of restarts we can take.

Symmetries in CSPs cause redundant work when using complete search. Subtrees of search can be considered equivalent under some symmetry. Whatever computation we exerted in proving such a subtree contains no solution, the same work must be done for all its symmetric equivalents. The same is not true for incomplete search however. Consider breaking symmetry by adding redundant constraints e.g. adding lex constraints to a matrix model. Though we are breaking symmetry, we are making the problem more constrained by adding to the constraints that must be satisfied. This leads to a smaller distribution of solutions, which in turn makes it harder for local search to find a solution [Pre01]. It was argued later that super symmetric models are preferable for incomplete search [Pre02].

A survey paper that discusses many different methods of complete search for CSPs was written by Dechter and Frost [DF99]. An introduction to search methods is presented in [Pre00] that covers some complete and incomplete search strategies.

2.2 Constraint Solvers

Research into constraint programming has yielded many different methods of solving CSPs. A constraint solver is an implementation of many of these propagation and search techniques, global constraints etc. so that CSPs can be solved efficiently. These implementations provide a clear method for constraint programmers to encode their problems.

There are many constraint solvers available today. Solver (produced by the company Ilog) is a C++ library which provides an imperative constraint solving toolkit. As well as containing many efficiently implemented global constraints, Solver works by maintaining consistency on constraints while using a simple backtracking algorithm.

ECL^iPS^e (produced by ic-parc, a company owned by Imperial College London) is a constraint logic programming system with Prolog-like syntax. These are the main two solvers used throughout this thesis. The constraint solver FreeSolver is introduced in Chapter 5. This solver is an imperative constraint solver written in Java which allows constraints of

arbitrary arity constructed using Java's arithmetic and logical operators. This solver was written specifically to perform the experiments found in Chapter 5

2.3 Computational Group Theory

In this thesis we use group theory widely, both as a means of describing symmetries but also as a means of computing with symmetries. However, the work carried out in this thesis mainly deals with computational group theory (CGT) as a tool to perform research into symmetry in the context of constraint programming. As such, we will mention some computational group theory software packages but not specifically the research that went into developing them. We will also mention some papers that make heavy use of group theory to solve combinatorial search problems, although not specifically using constraint programming techniques.

GAP [GAP03] is the main CGT system used in this thesis. It is an interpreted programming language that contains all the common functionality of an imperative language e.g. `for` and `while` loops, `if` / `else` conditionals, recursive functions and many arithmetic and logical operators. As well as this, there are thousands of library routines for solving common algebraic problems to do with numbers, graphs and various types of group. Another similar system is Magma [BCP97]. The work in [CF93] and the books [But91] and [Ser03] provide many informative descriptions and algorithms for permutation groups.

Soicher used the complex symmetries of SOMAs (mutually orthogonal latin squares) to good effect using GAP [Soi99]. Royle [Roy98] and McKay [McK98] present methods of enumerating "combinatorial objects" whereby only one representative is produced from each isomorphism class.

2.4 Symmetries in CSPs

The study of symmetries in CSPs has seen a large increase in interest over the past few years. Evidence of this can be shown by looking at the popularity of the "Symmetry in Constraints" workshop held at the annual "Principles and Practice of Constraint Programming" conference. The number of accepted papers rose from 11 in its first and second year

(2001-2002), to 18 in 2003.

Though before 2000, work into symmetries in CSPs was not as widespread, many fundamental papers were published which first highlighted the redundancy of constraint solvers dealing with symmetric problems.

In order to present a review of symmetry breaking in constraint programming we will partition previous work into two sections. The first section will look at research carried out in the 1990s. This work was generally concerned with breaking either specific types of symmetry, or breaking symmetries within a specific framework. The second section will look at papers from the 21^{st} century that mostly present general methods of breaking all symmetry. These papers will define symmetries to be as expressive as Definition 1.7 unlike most of the research discussed in the first section. Also, in the second section, we see more work which is a by-product of specific symmetry breaking methods i.e. papers which compare different symmetry breaking methods, or modify symmetry breaking methods etc.

Also of note, although outside the scope of this thesis, is the continuing interest of the study of symmetry in other areas of combinatorial search. We are seeing symmetry becoming a greater focus of study in SAT [Gol02] [LJP02] [ARMS03], planning [FL02], scheduling [Sel03] and integer programming [Mar03] to name just a few areas.

2.4.1 Symmetry Breaking in the 1990s

Thought by many to be the first work that looked at symmetries in constraint programming [Fre91], introduced the concept of neighbourhood interchangeability. In this paper, the problem of graph colouring was examined: given an undirected graph G, and k colours, a k colouring of G, is a labelling such that every node in G is associated with one of the k colours. No two nodes that share an adjoining edge may have the same colour. This problem has a great deal of symmetry since any solution can be transformed by performing any of the $k!$ permutations of the colours.

This paper however, dealt with a significantly weaker form of symmetry. Any two domain elements $a, b \in D(X_i)$ are said to be **fully interchangeable** iff any solution to the CSP which contains $X_i = a$ remains a solution if the assignment is changed to $X_i = b$ and vice versa.

In this case the symmetries are value symmetries i.e. only the domain elements are permuted. The symmetries are also only specific to one particular variable. Thus even if $\forall i, j \;\; D(X_i) = D(X_j)$, and a and b are fully interchangeable, we need to state explicitly which variable's domain a and b belong to.

The concept of value symmetry being global over all variables in a CSP is introduced in [Ben94]. Benhamou and Sais had already examined symmetries in propositional calculus earlier in [BS92]. In [Ben94], a system is developed for eliminating value symmetry in binary CSPs. A syntactical symmetry is defined as a permutation σ, such that, iff $\forall i, j \;\; (D(X_i)_a, D(X_j)_b) \in C_{ij}, (D(X_i)_{\sigma(a)}, D(X_j)_{\sigma(b)}) \in C_{ij}$. A filtering algorithm $revise(D(X_i), D(X_j))$ is used during search to remove symmetric nogoods.

This paper presents good empirical results on proving the insolubility of the pigeon hole problem: given n variables each with the domain from 1 to $n-1$, each variable must take a different value. The pigeon hole problem has freely interchangeable value symmetry. Though the technique in this paper applies only to binary CSPs, the author states that any non-binary CSP can be converted to a binary CSP [RPD90].

In [Pug93], Puget describes a symmetrical problem as one where, "some permutations of the variables map a solution onto another solution". This definition encapsulates the idea of symmetries acting on variables. Interestingly, this paper also defines symmetrical constraints. For example, the binary constraint $\neq$ is described as symmetrical since it is commutative. Similarly, so too is the global all-different constraint. Such symmetry is broken by lexicographically ordering the values of the variables in any one constraint. These 'lex' constraints are added to the problem. This method of dealing with symmetries, is still popular today and essentially transforms the original CSP into another CSP with less (or no) symmetry. One of the major disadvantages with using ordering constraints however is that different heuristics can have a detrimental affect [GHK02]. For example, consider including the following constraint to break symmetry: $X_1 < X_2 < ... < X_n$ with the value heuristic that tries to instantiate the largest value first. Trying to instantiate the variables from X_1 to X_n will result in many needless backtracks.

The paper by Crawford et al. [CGLR96] is a seminal paper in the field of symmetry in combinatorial search. Though the paper studies symmetry breaking with SAT, its implications are still relevant to CSPs.

Firstly, the paper extends the approach used in [Pug93] to break a symmetry by adding a lex constraint to the problem. Unlike Puget, the approach was automated i.e. given any symmetry, a relevant lex constraint could be constructed. Secondly, a technique from [Cra92] was used to automatically detect the symmetries of a given problem by using a graph automorphism algorithm. Thus, symmetries that resulted from permutations of variables and values could be found.

This was therefore the first paper that could break an entire set of arbitrary symmetries. However, an exponential number of lex constraints are needed to break all symmetries in general.

The final paper we will look at in this section is [MT01][2]. In some ways it is a step back from the approach in [CGLR96] since it does not break all symmetries, and the definition of symmetry used does not encapsulate all types of variable and value symmetry. A symmetry is described as a bijective mapping $\theta : X \rightarrow X$. This is enough to describe variable symmetries. The symmetry θ also contains another d bijective mappings (where d is the domain size) $\theta_1, ..., \theta_d$, where $\theta_i : D(X_j) \rightarrow D(X_j)$. This description allows many types of symmetry but not all. The work in [FM01] uses the same symmetry definition and thus has the same limitations.

Consider the n-queens problem (Model 2.3) where we have the symmetry: rotate the chess board by 180°. This is a symmetry that we can express by permuting the variables (for 4-queens, swap X_1 with X_4 and swap X_2 with X_3) and *at the same time* reversing the domains of all variables.

The rotate by 90°, rotate by 270°, and the flip about the diagonals are symmetries of this problem. However, they cannot be described using the definition of symmetry in [MT01]. This paper states that, "The remaining four symmetries of the chessboard are not symmetries of this formulation." This is not the case. These symmetries are in this model but they cannot be expressed using this method of describing symmetries. We need a more powerful method.

One of the major outcomes of this paper however is a symmetry breaking heuristic. This heuristic instantiates the variable that has the most symmetries acting on it at that node in search. More importantly, the implementation saw a return to a dynamic symmetry break-

[2]The original version of the paper first appeared in IJCAI 1999.

ing method i.e. like [Ben94], it is used during search. In a similar manner to Benhamou, this method of symmetry breaking removes domain elements that are symmetric to previously found nogoods. The difference here is that the symmetries are more complex than just value symmetries.

2.4.2 Symmetry Breaking in the 21^{st} century

The second section of papers on symmetry breaking in constraint programming, deals mainly with dynamic methods used during search, and general methods for breaking arbitrary symmetries.

The first of these methods is that developed by Backofen and Will [BW99] and later by Gent and Smith [GS00]. The method, named SET (Symmetry Excluding Trees) in [BW99] and SBDS (Symmetry Breaking During Search) in [GS00] are essentially the same except for one optimisation in SBDS. Once a symmetry can be shown to be *broken* in the current subtree of search, it is discarded from consideration. Whenever we refer to SBDS in this thesis, we are referring to the two approaches collectively.

SBDS, like the method introduced by Crawford et al. [CGLR96], can break all symmetries in a CSP. The symmetry breaking functions also allow all possible variable and value symmetries. One area in which SBDS is superior to previous methods is that it does not conflict with the heuristic being used. All symmetric variants of nogoods are forbidden, so it does not matter in what order the search is traversed.

To break symmetry using SBDS, the constraint programmer supplies a list of functions to represent the symmetries of the CSP. While searching for a solution, we record the set of decisions made as the state A. Upon backtracking from a failed assignment: $X_i = a$ we add the following constraint to the local subtree for every symmetry function g.

$$A \wedge g(A) \wedge (X_i = a) \Rightarrow g(X_i = a)$$

The optimisation mentioned above in the method by Gent and Smith is that, once $g(A)$ is guaranteed to be false in a subtree, we discard g from consideration in that subtree. The SBDS method can be seen as an extension to [MT01] where non-unary constraints are also posted. SBDS has been successfully used in many different papers [Smi01] [FSS01]

[FM01] to break over 10^4 symmetries [MS02] and improve run-times by many factors [GLS00].

One problem with SBDS was the redundancy of duplicate constraints. For example, two different symmetry breaking functions used by SBDS g and h may be used to construct the same constraint. If a CSP had more than $n \times d$ symmetry breaking functions for a problem with n variables with domain size d, then backtracking to the root node of search would result in posting duplicate constraints.

A follow up to [GS00] by Gent, Harvey and Kelsey [GHK02] contains a modification to the SBDS implementation which uses group theory to deal with larger groups in a tractable way. The GHK-SBDS symmetry breaking method uses group theory to avoid posting many of the duplicate constraints that SBDS did.

Previous methods by Puget, Crawford et al., Backofen and Will, and Gent and Smith all add constraints to the solver. This is done either as a pre-processing step before search commences or they are dynamically added to a subtree in search. The following approaches are used dynamically during search but rather than add constraints to the solver, they inform the solver when to backtrack in search.

Surprisingly, this approach was first used in 1988 by Brown, Finkelstein and Purdom [BFP88] [BFP96]. Though their approach was not specifically for CSPs but backtrack search, the ideas remain applicable to constraint programming. Given a group representing the symmetries of a problem, if a group element could be found that mapped the current state in search to a lexicographically smaller one, we should backtrack. One restriction that this method entailed is that we must search for a solution using the static lexicographically least ordering heuristic.

A similar approach was developed for the CSP framework, and for any heuristic, independently from each other (and from Brown et al.) by Fahle, Schamberger and Sellmann [FSS01] and Focacci and Milano [FM01]. Similar to SET and SBDS, the technique will be collectively referred to as SBDD. Given a state in search P^c and a set of nogoods $\mathcal{T}$, we look for a symmetry that maps any element of $\mathcal{T}$ to a subset of the decisions in P^c. If such a symmetry can be found, then we have entered a node in search symmetrically equivalent to a previously visited nogood and therefore, we can backtrack. The test to find such a symmetry is called a "dominance check" because we are checking to see whether

our current state in search is dominated by any of the other previously failed states. SBDD contains an optimisation to limit the exponential number of failed states to consider, to at most $(n-1) \times (d-1)$. The dominance check, itself encoded as another CSP, is equivalent to the NP-complete problem of subgraph isomorphism. The technique has been used to break all symmetries in problems with over 10^7 symmetries.

A modification to SBDD was proposed by Harvey [Har01] which contains two notable improvements. Firstly, a method of uniting all previous failed search states into one state is presented. Secondly, rather than mapping the set of all domains, we look at just the set of decisions made. This makes the set of points acted on much smaller. These two changes reduce the amount of computation needed to use the symmetry breaking method SBDD. When we refer to SBDD, we collectively consider the original SBDD technique with the improvements introduced by Harvey.

The GHKL-SBDD method by Gent et al. [GHKL03] similarly uses the same idea as SBDD: backtrack whenever we enter symmetrically redundant search. However, whereas SBDD insists the constraint programmer writes their own dominance checker for each problem, GHKL-SBDD, like GHK-SBDS, only requires a group representing the symmetries of the CSP.

As well as research that introduces new methods of breaking symmetry, we have seen many examples of using symmetry breaking methods to solve problems with symmetry effectively [Har01] [Pug02] [Pea03] [PS03]. Much of this work notes small improvements that can be made to the symmetry breaking method in general or when dealing with specific problems.

It is clear that the current research into symmetry in constraint programming is far reaching and abundant. We are seeing more research now than ever before into the various aspects of symmetry breaking: new methods, experiments, modifications to methods for specific problems, solving specific symmetrical problems, implementing symmetry breaking systems, combining methods, and creating efficient methods of breaking specific types of symmetry.

Chapter 3

Implementation of Symmetry Breaking Systems

A constraint solver is made of many parts [Kum92]. There are many different ways to traverse the search space of a CSP: backtrack search, conflict directed backjumping [Pro93], limited discrepancy search [HG95], depth bounded search etc. There are also many levels of consistency to enforce and even more different ways to accomplish them [Mac77] [Coo89]. Finally there are many popular dynamic variable and value ordering heuristics we can choose: most constrained first, smallest domain first or kappa [GMP+96] [Smi97] etc.

At present there are also a large number of different methods of breaking symmetry in CSPs. A symmetry breaking system is quite simply an implementation of one or more symmetry breaking methods that can be used to avoid redundant work performed by the constraint solver.

In the future we hope to see symmetry breaking systems playing an equal role to search and inference techniques and heuristics for symmetric problems in constraint solvers. This chapter will be looking at the importance of implementing symmetry breaking methods in terms of efficiency, ease of use and generality. We will look at the consequences of implementations of symmetry breaking systems throughout the chapter by using three symmetry breaking systems as examples: SBDS [GS00][BW99], SBDD [FSS01][FM01] and GHK-SBDS [GHK02]. By identifying what the benefits and disadvantages of certain symmetry

breaking systems are we can make superior implementations which will help to take symmetry breaking research into the mainstream of constraint solvers.

3.1 Requirements of a Symmetry Breaking System

As was reviewed in the previous chapter, there are many methods for breaking symmetry in CSPs. At the moment, these methods are used by researchers working on symmetry in CSPs. These researchers will have a detailed knowledge of the specific symmetry breaking method most suited to them. If we are to look forward to more symmetry breaking systems to be included in constraint solvers, we need to look at different symmetry breaking systems and see how they differ. We also need to examine the strengths and weaknesses of these symmetry breaking systems.

3.1.1 Automatic Symmetry Detection vs Symmetry Descriptions

One important task that needs to be performed before symmetry breaking can occur is the process of actually identifying the symmetries. A symmetry breaking system needs to decide whether identifying the symmetries of a CSP is a job for the constraint solver or the constraint programmer. The most popular choice to date is that the constraint programmer describes the symmetries of the problem. However, both choices have advantages and disadvantages that will be considered in this section.

Some symmetry breaking systems break only a specific type of symmetry and thus no symmetry needs to be detected or described. Such systems can only be used if the specific symmetry exists. For example, lexicographic ordering constraints can be added to matrix models with interchangeable rows and columns to break some symmetry [FFH+02]. Also, the STAB method has currently only been used to break such symmetries [Pug03]. Freely permutable value symmetry was examined in [Gen01] and [HFPA03].

Graph automorphism and isomorphism detection is used in data flow diagrams to find symmetries in Digital Signal Processing problems [vEJMT99]. Both [CGLR96] and [JR97] use a graph automorphism algorithm to find symmetries in CSPs, a technique introduced in [Cra92]. Finding symmetries in SAT has also been accomplished by using a graph au-

tomorphism algorithm [ARMS03]. Special cases using boolean variables introduced their own methods of detecting symmetries [Agu93] [BS92]. Most modern symmetry breaking systems in the field of non binary constraint satisfaction however, rely on the fact that spotting symmetries in CSPs is generally easy. The systems developed in [MT01] [BW99] [GS00] [FSS01] [FM01] all assume the constraint programmer describes the symmetries.

The latter papers mentioned above specify that the constraint programmers must supply the symmetries of the problem. By doing so, the work of both identifying the symmetries and encoding the symmetries are passed to the constraint programmer. Since constraint programming allows for very expressive models and easy problem representation, it is a general assumption among the symmetry breaking community that recognising a great deal, if not all, of the symmetry in ones own programs is generally straightforward. Once the symmetries have been recognised though, they must then be encoded somehow. We can see this as an extension of the encoding of a CSP - we must encode the model, the constraints and now we must also encode the symmetries.

In effect, this adds an extra layer of code that needs to be written, the format of which is specified by the symmetry breaking system. Even though it may be trivial to recognise the symmetries of a given CSP, how easy it is to encode these symmetries is entirely dependent on this *format*. This fact is extremely important. The method of encoding symmetries for a given symmetry breaking system must allow us to represent any symmetries that may occur and we also need to be able to do this in an easy, straightforward manner.

At present, the ECL^iPS^e constraint logic programming system [WNS97] is the only constraint solver that contains a symmetry breaking system: an SBDS library. Symmetry breaking is still largely in the domain of researchers. Since researchers will most likely have their own specific implementation, ease of use is not usually a large concern. This raises the important issue that a symmetry breaking system to be used by the average constraint programmer must either detect symmetries automatically or supply well documented, easy to use and expressive methods of *describing* the symmetries.

We will now look at symmetry breaking systems that automatically detect the symmetries of a CSP. The general method of detecting symmetries automatically is to convert a CSP into a graph. We then solve the *graph automorphism* problem on this graph which gives us the generators of group representing the symmetries of the constraint graph. The complexity of performing graph automorphism is equivalent to that of graph isomorphism which

is known to be hard. This problem is not classified as being either NP-complete or in P [JR97], however it can be solved by group theoretic algorithms that work well in practice. In general, a large number of generators are returned by the graph automorphism algorithm and some important group theoretic operations have complexities that depend on the size of the generator set e.g. the complexity of the orbit finding algorithm is $\mathcal{O}(|k| \times |o|)$ where k is the generator set and o is the orbit.

One limitation of finding automorphisms of the constraint graph is that only symmetries of the constraints are found. There may be constraints that are not represented since they are implicit in the model. Since these constraints are not listed, the constraint graph does not represent all the symmetries that exist in the problem. As an example, consider Model 2.3 of the n-queens problem. We have an *all different* constraint on the variables that ensures each queen is placed on a different column. We do not need such a constraint to ensure only one queen must be placed on each row. This is because each row is represented by one variable and thus it can only take one value. Therefore the symmetry where the board is rotated by 270° or by 90° is not detected by the graph automorphism algorithm.

CSP encodings can be very concise due to the expressive power of constraint programming. SAT encodings however, are very opaque and thus detecting symmetries by using the graph automorphism algorithm is an easier method of describing symmetries in SAT than produces generators manually [ARMS03].

One interesting approach to symmetry breaking used in SAT solving but not in CSP solving is the idea of using the graph automorphism check at every node. Once the symmetries of a problem G, have been detected, it is generally assumed that any symmetry acting on a state during search will be a subset of G. However it is possible that after making decisions, rather than reducing the number of symmetries that are still valid, some more are created [GMS03]. By using the graph automorphism algorithm at every node, we can try to detect symmetries that are introduced to the problem during search.

If a symmetry breaking system has automatic detection it is preferable to detect all the symmetries that exist in the problem and should run in a reasonable time. A symmetry breaking system that requires the constraint programmer to describe the symmetries should provide a language that is expressive enough and straightforward and easy to use.

3.1.2 Expressiveness

The definition of symmetry used by the symmetry breaking system affects its expressiveness i.e. the limitations of which symmetries it is possible to represent. As was shown in Chapter 2.4, if the definition of symmetry is not expressive enough then it may not be possible to break all symmetry [MT01] [FM01]. Some symmetry breaking systems only allow specific types of symmetries to be expressed e.g. lexicographic ordering constraints used with matrix models can only break permutation symmetries. Whether or not all permutation symmetries are broken by lexicographic ordering on a matrix model is not the issue here but rather, can permutation symmetries be expressed?

Expressiveness is an important factor for symmetry breaking systems that require the user to describe their own symmetries. As was mentioned above (Chapter 3.1.1), a symmetry breaking system that does not detect symmetries must supply a means of describing symmetries. The ideal symmetry breaking system should have an expressive language in which to describe symmetries.

Definition 1.7 describes a symmetry as a bijective function i.e. we can represent a symmetry as a function that takes a parameter and returns a result. Choosing the parameter to be a set of assignments is more expressive than choosing the parameter to be a set of variables and allows more symmetries to be represented. By making symmetries that take a state during search e.g. a set of domains of all the variables, and return its symmetrically equivalent state, this allows the most expressive symmetries. It should be noted that while some symmetries may only need to be described in terms of variables, in general there are a lot of symmetries found in CSPs that apply to combinations of variables and values. In the cases where symmetries *act* on just variables or just values, it may be beneficial to the symmetry breaking system to *represent* symmetries as just acting on variables or values. However, it is necessary to ensure that if a CSP has symmetries acting on assignments, then the symmetry breaking system should be able to express them.

The SBDS functions in [GS00] essentially take an assignment and return the symmetric assignment relevant to the specific SBDS function. The SBDD dominance check found in [FSS01] takes a state in search i.e. the current set of domains of all variables, and applies symmetries to it to try to find a superset of an already failed state.

As mentioned in Chapter 1.3.1, any group can be represented as a permutation group, and

every permutation group acts on a number of points. For GHK-SBDS, the points refer to what the symmetries act on e.g. if a CSP has n variables which are symmetrically equivalent, we can construct a group acting on n points representing the symmetries of this problem. If a CSP with n variables, each of which has a domain of size d, has some symmetries acting on the assignments, we need a group acting on $n \times d$ points to represent the symmetries. For GHK-SBDS, it is the responsibility of the user to select an appropriate number of points for the group to act on. It is this number that affects the expressiveness of the symmetries. Even though GHKL-SBDD [GHKL03] has a completely different symmetry breaking technique, the symmetry breaking representation is identical[1]. All the above 4 methods have an expressive enough representation of symmetry to describe any symmetry as defined by Definition 1.7.

3.1.3 Symmetry Representation

The symmetry representation of a symmetry breaking system is concerned with the way symmetries are stored internally. Generally speaking, most current symmetry breaking systems store symmetries as a group or something equivalent to a group i.e. a set of symmetries that can generate any symmetry of the problem.

The benefits of using just group generators as a symmetry representation include small memory requirements since the size of the generator sets is usually quite small, and they allow the symmetry breaking system to make use of group theory techniques i.e. if the generator set of symmetries is consistent with the four axioms of a group, then computational group theory and the large literature on group theory and can be used.

The other main representation of symmetry in a symmetry breaking system is as a list of symmetries. Storing symmetries as a list generally leads to simpler implementations however, the symmetry breaking system is quickly limited to small number of symmetries. The largest number of symmetries used with such a system is less than 10^5 symmetries [MS02].

SBDS represents the symmetries of a CSP as a list. Each symmetry is represented as a function and the symmetry breaking is performed by operating on the list of these functions. GHK-SBDS represents the symmetries of a CSP as a group and makes use of the

[1] One minor exception is that the GHK-SBDD group has to act on all $n \times d$ points.

efficient group theory algorithms of GAP [GAP03]. Although it is less clear to see how the symmetries are represented by SBDD, it is as a set of symmetries that can recreate all the symmetries of the CSP, essentially a generator set of a group. The SBDD dominance check contains "state transitions" that take a state[2] in search A, and map it to a symmetrically equivalent state A'. Each state transition is a symmetry of the problem. When SBDD tries to find a dominating state, the dominance check is trying to find a combination of state transitions that map a previously failed state to a subset of the current state. By finding such a mapping, the dominance check has found a symmetry that should be broken by SBDD. Even though this symmetry was not represented internally, it was found by combinations of the generator set of symmetries.

3.1.4 Problem Specific or Instance Specific

One great advantage of constraint programming is that we can describe the model and constraints of the problem without needing to explicitly know the size of the problem i.e. the number of variables and/or the size of their domains. For many constraint problems it is possible to represent the number of variables as a constant variable[3] n, and refer to n when required. We can then state the *size* of the CSP at run-time and not compile-time thus making it easy to write general programs to solve constraint problems that vary in size. For example, consider the n-queens problem. This is a class of problems with specific instances such as the 4-queens problem, and the 8-queens problem etc.

If a symmetry breaking system is to be included in a constraint solver, it must be able to refer to the size of the CSP (without needing to explicitly know it) and still be able to break symmetry at run-time i.e. be able to deal with *problem* specific symmetry breaking and not just *instance* specific symmetry breaking. There are two main issues to deal with if this problem is to be solved. Firstly, even if the symmetries of a problem do not change with its size, their internal representation may do. If we look at the group representing the symmetry acting on assignments of the 2-queens problem using Model 2.3, the generator set is $(1,2)(3,4), (1,2,4,3)$. The generator set for the 3-queens problem is $(1,3)(4,6)(7,9), (1,3,9,7)(2,6,8,4)$. It is clear the group representation will change with size since the number of points they act on will vary.

[2] In [FSS01], the domains of each of the variables.

[3] Not as a constrained variable.

Secondly, many symmetric CSPs have a varying amount of symmetry with respect to their size. A CSP with n variables that are symmetrically equivalent to each other, will have $n!$ symmetries. A CSP that is modelled on a square will have 8 symmetries regardless of the size of the square. A symmetry breaking system needs to cope with varying numbers of symmetries at run-time.

Since SBDS takes a list of symmetry breaking functions or predicates, they need to be produced at compile time for non-interpreted languages such as Solver. It is possible for these functions to work with CSPs of various sizes with constant amounts of symmetry in certain cases e.g. n-queens problem in [GS00]. However, since these functions are written as code that needs to be compiled to work, SBDS cannot cope with CSPs whose number of symmetries vary with their size. CSPs that have a varying amount of symmetry with respect to their size need to run a separate program to output and compile specific functions for a specific size of CSP if they are to use SBDS. This is not an ideal solution as it involves a considerable amount of extra coding for the constraint programmer.

The SBDD dominance check has an internal representation of the generator set of the group representing the symmetries of the CSP. The size of the set can remain constant and still vary the size of the group by varying the size of the set of points the group is acting on.

The implementation of GHK-SBDS in [GHK02] leaves the constraint programmer to write GAP code to describe the group. The constraint programmer can write GAP code to describe a specific permutation group by listing a generator set of permutations explicitly i.e. `g := Group((1, 2), (3, 4));`. By doing so, the constraint programmer limits their symmetry breaking to be instance specific. If the constraint programmer wishes to break problem specific symmetry, they must write generic GAP code that produces a group based on the size of the CSP. Since GAP is a language complete with all the necessary programming constructs (`if` statements, `for` loops etc.) and group theory library functions (`g := SymmetricGroup(5)` etc.), it is possible to write such generic code. The constraint programmer may need to learn more group theory and/or GAP in order to achieve this.

3.1.5 Breaking all Symmetry

Symmetry breaking can be used to limit the number of solutions returned by a constraint solver for a problem with loose constraints. It can also be used to reduce run-times. Some

constraint programmers may wish to know the exact number of unique solutions. If the latter is required than a symmetry breaking system that guarantees to break all symmetry must be used.

Lexicographically ordered matrix models do not break all symmetry and cannot guarantee unique solutions. SBDS, SBDD and GHK-SBDS do however break all symmetry and return unique solutions. The constraint programmer must be aware though that there is a performance cost in applying symmetry breaking that increases with respect to the size of the group representing the symmetries of a CSP. If the size of this group is too large, the symmetry breaking systems above may not be able to operate efficiently enough i.e. there is a limit to how much symmetry can be efficiently broken. In such cases where there are too many symmetries, a subset of all of them must be broken. The topic of breaking subsets of symmetries is the area of study of Chapter 4.

In general though, a symmetry breaking system should ideally break all symmetry but if not, it should be able to deal with large amounts of symmetries efficiently by breaking just a subset of them.

3.1.6 Ease of use

Perhaps the most important feature from the point of view of the constraint programmer ignorant of symmetry breaking techniques is the ease of use of the symmetry breaking system. Constraint programmers do not have to know anything of the implementation of arc-consistency algorithms in order to use them. Similarly, constraint programmers should not need to know anything about symmetry breaking techniques in order to use them. Therefore, the interface between using a symmetry breaking system and the constraint programmer should be as easy to use as possible.

There are many features of modern symmetry breaking systems that may discourage the average constraint programmer from using them. For example, constraint programmers may not wish to write more functions (or a program to write more functions) as is needed by SBDS. The constraint programmer may not wish to write a new dominance check for every CSP as is required by SBDD. Finally, they may not wish to learn the group theory needed to use GHK-SBDS or GHKL-SBDD. Due care must also be taken when using GHK-SBDS that the labelling of the assignments to points and vice versa is consistent with

the GHK-SBDS implementation. This list is of course subjective and will vary from user to user but, the easier to use the symmetry breaking system is, the more likely it is to be accepted by constraint programmers in general.

3.1.7 Combinations of methods

Some symmetry breaking methods are mutually exclusive such as SBDD and SBDS in that they both prune redundant search so only one technique is required. Some methods have consequences that need to be considered before using other techniques. Imposing static symmetry breaking constraints changes the symmetries of the problem which could have adverse affects if not considered carefully. For example consider a problem where we have n variables: $v_1, v_2, ..., v_n$ that are all symmetrically equivalent. If we post a symmetry breaking constraint to order these variables lexicographically ($v_1 \leq v_2 \leq ... \leq v_n$) we cannot use SBDS to break the permutation symmetry. Consider searching for a solution to this problem and a partial assignment $v_1 = 4 \wedge v_2 = 1$ is reached. We would fail from this point due to violating the ordering constraint. If we were using SBDS then upon backtracking, constraints could be posted to rule out the partial assignment $v_1 = 1 \wedge v_2 = 4$. However, this could be a valid partial assignment.

Puget describes a way of combining symmetry breaking systems and lexicographic ordering constraints in [Pug03] which relies on the variable and value ordering heuristics being compatible. Some methods can be combined thus giving greater benefits, such as using lexicographic ordering constraints to break some symmetry and using SBDS to break any remaining. When doing so however, due care needs to be taken to ensure the same symmetry is not broken in incompatible ways.

3.2 Unique Symmetry Breaking using Group Theory

This section details the creation of a symmetry breaking system written for use with Ilog Solver 4.4. It is the first system since [BFP96] to use group theory techniques to break symmetry in constraint programming. The group theory representation and algorithms were written in C++. This symmetry breaking system also discards non-unique symmetries thus reducing the number of symmetries to consider.

This symmetry breaking system, Unique Symmetry Breaking During Search (henceforth U-SBDS) is based on SBDS. When backtracking from a failure, we post constraints in the current subtree that forbid any symmetrically equivalent state. There are 3 main differences between this symmetry breaking system and SBDS. Firstly, since group theory techniques are being used the constraint programmer does not have to write a list of functions but merely the generators of the group. Secondly, unlike SBDS which has a list of the symmetries, this new symmetry breaking system cannot keep track of which symmetries have been broken. This leads U-SBDS to post constraints forbidding states that can never be reached from the current subtree. Thirdly, one disadvantage of SBDS, and to a lesser extent GHK-SBDS, is that it posts redundant identical constraints. Given a set of symmetry breaking functions, some states can be mapped to the same different state by many of these symmetry breaking functions. U-SBDS overcomes this by using the group theory setwise orbit finding algorithm which finds the set of distinct states that can be reached by the group representing the symmetries of the problem.

A consequence of using the orbit finding algorithm to find symmetrically equivalent states during search is that the constraint that is added to the solver is calculated differently to SBDS. After failing during search with assignment $var = val$, and backtracking to state A, we add this constraint to the current subtree:

$$g(A) \quad \Rightarrow \quad g(var \neq val) \tag{3.1}$$

When using U-SBDS however, we say that $\{A \bigcup var = val\}$ is the state that failed and we calculate (and forbid) the orbit of that set of decisions. The resulting output is a set of sets and as such, the position of $g(var \neq val)$ will most likely not be the last element of the set. We can no longer post a constraint of the form $A \Rightarrow \neg B$. It is possible, however, to expand Equation 3.1:

$$\begin{aligned}
g(A) &\Rightarrow g(var \neq val) \\
\neg g(A) &\vee g(var \neq val) \\
\neg g(v_1 = val_a \wedge ... \wedge v_k = val_x) &\vee g(var \neq val) \\
\neg g(v_1 = val_a) \vee ... \vee \neg g(v_k = val_x) &\vee g(var \neq val) \\
g(v_1 \neq val_a) \vee ... \vee g(v_k \neq val_x) &\vee g(var \neq val)
\end{aligned}$$

We can now post symmetry breaking constraints of the form:

$$\begin{aligned} \forall\{o_1, o_2, ..., o_k\} \quad &\in \quad Orbit(G, (\{a_1, a_2, ..., a_{k-1}\} \bigcup \{var = val\})) \\ \neg o_1 \vee \neg o_2 \vee \quad &... \quad \vee \neg o_k \end{aligned}$$

Where $Orbit(G, A)$ is the set of orbits of the state A under the group G and $\{a_1, a_2, ..., a_{k-1}\}$ is the set of choices made thus far i.e. the root node of the current state in search.

3.2.1 Unique Symmetries

After every backtrack we post symmetry breaking constraints. It would be perfectly valid to use the generators to recreate all the elements of the group and post a constraint for every symmetry. This would incur a very high overhead by flooding the constraint solver with too many constraints. At every node in the search tree some of the symmetrical constraints are the same as each other. In many cases there are more duplicate constraints than distinct ones. In order to reduce the overhead of posting duplicate constraints, we must ensure that at each node in the search tree we only consider the unique symmetries.

Definition 3.1 *Given a partial assignment A, the maximal set of symmetries G' is a subset of all the elements in G, is unique with respect to A iff* $\forall g \in G'(\neg\exists h \in G' | h(A) = g(A) \wedge h \neq g)$.

The maximal set of unique symmetries is equivalent to the orbit of the *set* of decisions made so far. Figure 3.1 contains the average size of the set of orbits for the 4×4 alien tiles problem (see Appendix C.1) with respect to the depth of the search tree. This problem has 1,152 symmetries.

3.2.2 Theoretical Analysis and Bound on Symmetry Breaking Constraints Needed

We can use the idea of unique symmetries to create an upper bound on the maximum number of symmetries breaking constraints after any one failure needed to break all symmetry.

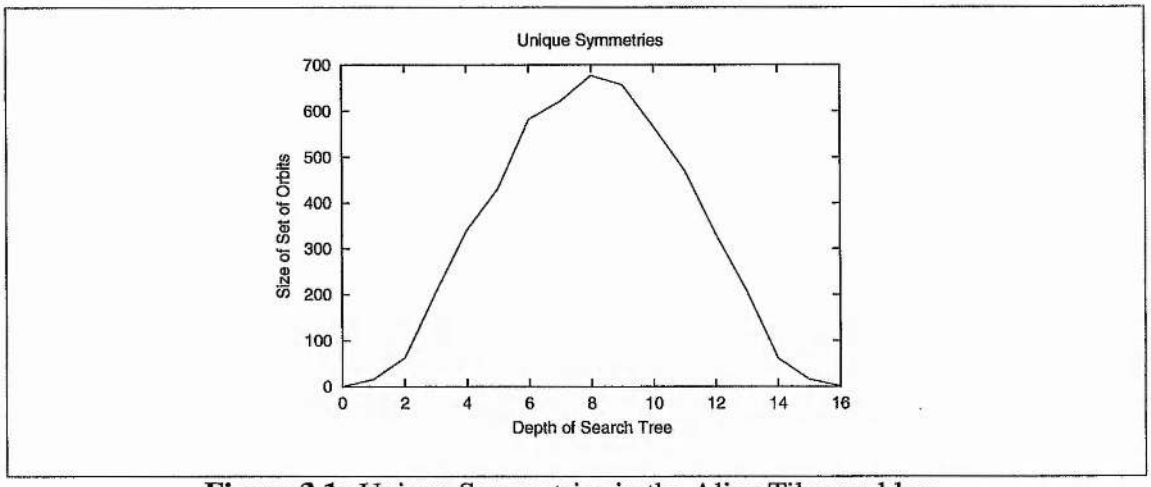


Figure 3.1: Unique Symmetries in the Alien Tiles problem.

The example of the alien tiles problems has $2n!^2$ symmetries for n^2 variables. For a group G acting on n points, the largest possible size of G is $n!$, the symmetric group. This is due to the fact that a symmetry has to be a bijection. Therefore, any upper bound for a group with $n!$ symmetries will be an upper bound for *all* groups. We will be looking at the symmetric group acting on just variables since the symmetric group acting on assignments yields a problem that has either no solutions or d^n solutions. Any group that has symmetries acting on assignments should replace the term n with nd to produce a correct upper bound.

We now propose to look at the number of unique symmetries for a larger group than that acting on the alien tiles problem: the symmetric group acting on ten points. We assume that this group, S_{10}, represents the symmetries acting on the variables of some imaginary CSP with 10 variables. The size of S_{10} is $3,628,800$. Figure 3.2 contains the number of unique constraints needed, plotted against the size of the failed partial assignment. The number of unique symmetries is the same for all nogoods of size k. This is true because the group we are using is the symmetric group. If G is the symmetric group then for any set of points α, β of size k: $Orbit(G, \alpha) = Orbit(G, \beta)$. Therefore, we only need to consider one set of points for each size of set.

Consider a failed partial assignment, or nogood, with 3 variables. The number of unique symmetries i.e. the upper bound for the number of symmetry breaking constraints that we need to post for this nogood is 120. This is over 30,000 times less than the entire group. This number will in general be much smaller as we only need to produce symmetry breaking constraints for the intersection of non-broken symmetries and unique symmetries.

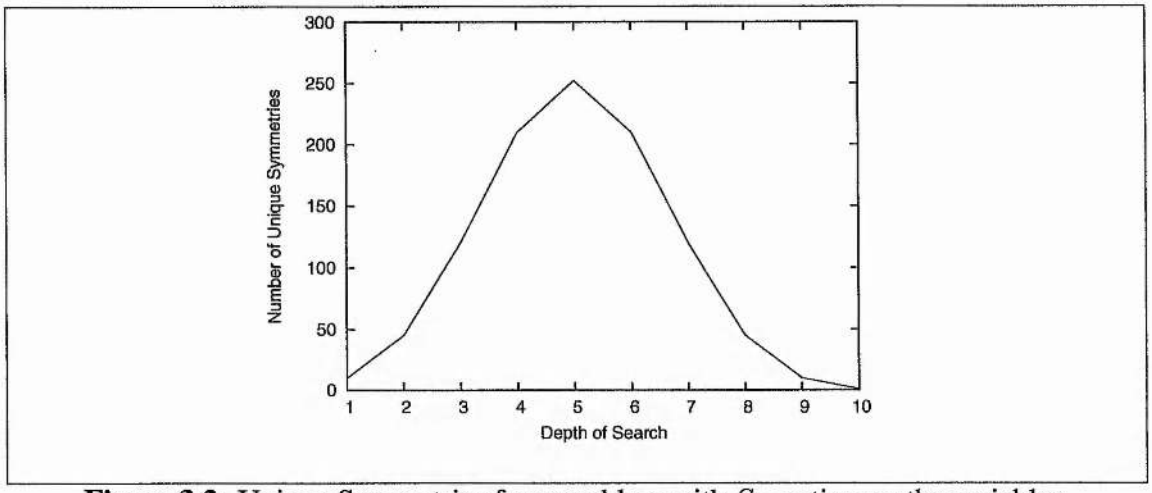


Figure 3.2: Unique Symmetries for a problem with S_{10} acting on the variables.

By examining Figure 3.2 we can infer two things. Firstly, the number of symmetry breaking constraints that need to be posted, is greatest halfway down the search tree. Secondly, and most importantly, the maximum number of symmetry breaking constraints needed at any time is a fraction of the entire set of all possible symmetries.

Note that for the symmetric group acting on n points, the size of the setwise orbit of a set of m points is equal to nCm. The nCm function (pronounced "n choose m") explicitly is:

$$\frac{n!}{m!(n-m)!}$$

The size of the pointwise orbit of a set of m points contains also contains all $m!$ reorderings, and thus is equal to:

$$\frac{n!}{(n-m)!}$$

Since the setwise orbit is largest halfway down the search tree, or more specifically when $m = \frac{n}{2}$, we have the upper bound of the maximum number of symmetry breaking constraints needed after any backtrack for any group acting on n points:

$$\frac{n!}{(\frac{n}{2})!^2}$$

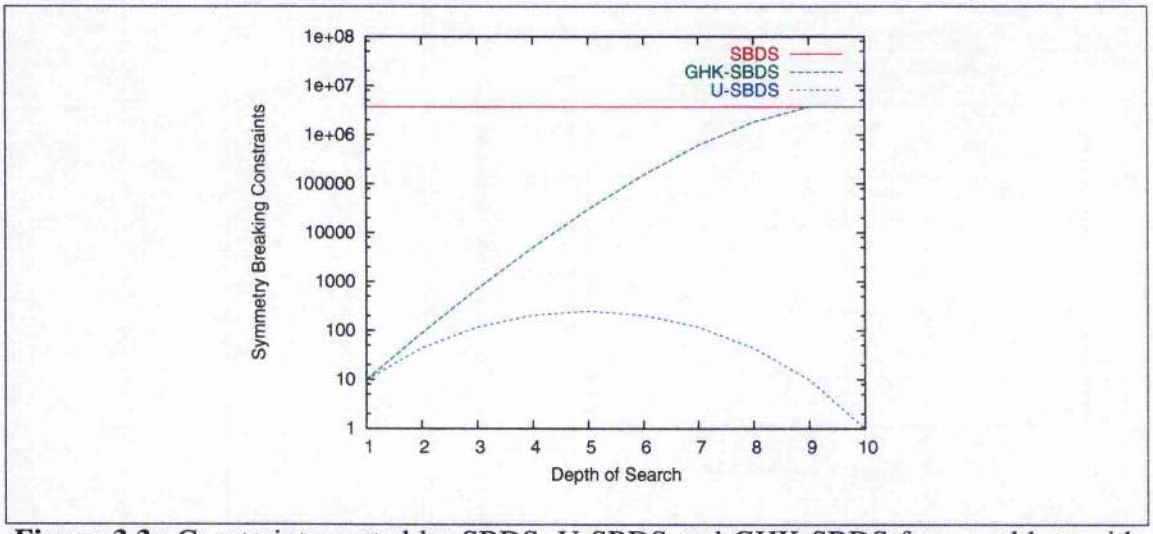


Figure 3.3: Constraints posted by SBDS, U-SBDS and GHK-SBDS for a problem with S_{10} acting on the variables.

Theoretical Comparison of SBDS and U-SBDS

The major disadvantage of SBDS is that at the root, all symmetries are considered. Thus if we backtrack to the root of search, we will post as many constraints as there are symmetries. For this reason, the number of symmetries that SBDS can effectively cope with is minimised. By breaking just the unique symmetries (as U-SBDS does) we can greatly reduce the number of constraints needed by similarly backtracking to the root node. However, whereas the large overhead of SBDS decreases as we traverse deeper into the search tree, the overhead of U-SBDS increases as we approach the depth halfway down the search tree.

Since there are exponentially more nodes of the latter type, we would expect U-SBDS to perform worse than SBDS. However, the size of the overheads of SBDS and U-SBDS are quite different. The largest number of constraints posted using SBDS to break the symmetries of S_{10} is $3,628,800$. As shown in Figure 3.2, the largest number of constraints posted using U-SBDS is 252. The size of the largest setwise orbit will always be much less than the size of the entire set of symmetries.

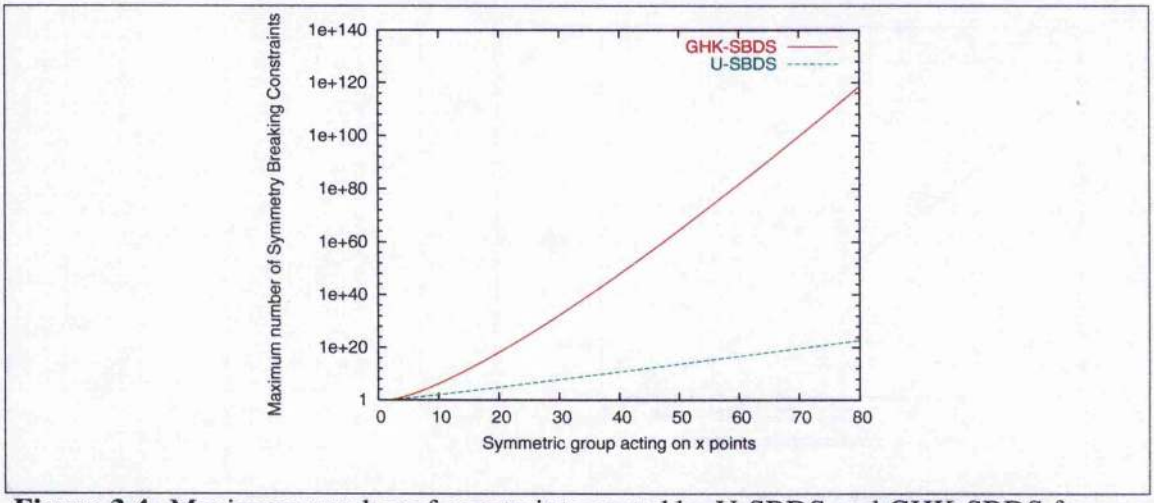


Figure 3.4: Maximum number of constraints posted by U-SBDS and GHK-SBDS for any one nogood from a problem with S_n acting on the variables for varying n.

Constraints posted by SBDS, U-SBDS and GHK-SBDS

Figure 3.3 contains an upper bound for the number of constraints posted by SBDS, U-SBDS and GHK-SBDS after a backtrack at a certain depth for a problem with S_{10} acting on 10 variables. Though the results of this graph appear to show U-SBDS posts the fewest symmetry breaking constraints, note that in practice SBDS and GHK-SBDS post far fewer as they do not consider guaranteed broken symmetries unlike U-SBDS. We can see however that there is a clear difference between using the setwise orbit to produce constraints (as U-SBDS does) and the pointwise orbit (as GHK-SBDS does). Recall that the pointwise orbit essentially allows the points in a set to be re-ordered. Whereas the setwise orbit of $\{1, 2\}$ and S_3 would be $\{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$, the pointwise orbit would be $\{[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]\}$.

Though a symmetry breaking system that breaks the intersection of unique symmetries and non-broken symmetries does not exist, it is conceivable that one could be constructed. Such a symmetry breaking system would theoretically be able to deal with larger groups than GHK-SBDS.

We now examine how the maximum number of symmetry breaking constraints required from any one nogood, increases with S_n for various n. Figure 3.4 shows the largest number of constraints needed from breaking all symmetries on S_2 to S_{80}, which has over 10^{118} elements. We compare the largest possible size of the setwise orbit (U-SBDS) and the

8-queens	Backtracks	Runtime	Solutions
Solver 4.4	324	0.06	92
U-SBDS	61	0.02	12
SBDS	61	0.01	12

Table 3.1: Results of finding all solutions to the 8-queens problem using different symmetry breaking systems.

pointwise orbit (GHK-SBDS). The increase in the setwise orbit is much less than the pointwise orbit. We can therefore use dynamic symmetry breaking constraints to break much larger groups than is reported in [GHK02].

Unfortunately, though the setwise orbit increases much slower than the pointwise orbit, it still increases at an exponential rate. This further enforces the theory that breaking an exponential number of symmetries in general, with the exception of value symmetry [RDGKL04], requires an exponential amount of computation.

3.2.3 Empirical Results

We now present some empirical results from solving symmetry problems using U-SBDS with Ilog Solver 4.4. Results are compared against using no symmetry breaking and SBDS. Table 3.1 contains the data from finding all solutions to the 8-queens problem. Table 3.2 contains the data from finding an optimal solution to the alien tiles problem with a 4×4 board with 3 colours.

Since the empirical benchmarks have a small number of symmetries, SBDS can easily cope with them. The largest number of symmetries that SBDS has reportedly used is 8,000 (see Chapter 4.5.3 and [MS02]). We have shown in Figure 3.2 that a large group such as S_{10} can be broken by posting a relatively small number of symmetry breaking constraints. Therefore, it is possible that U-SBDS may be able to solve problems that have too much symmetry for SBDS to cope with.

	Solver 4.4		SBDS		U-SBDS	
Cost	Fails	Runtime	Fails	Runtime	Fails	Runtime
1	0	0.37	0	3.74	0	0.42
2	0	0.38	0	3.77	0	0.42
3	0	0.39	0	3.95	0	0.44
4	0	0.40	0	3.99	0	0.46
5	0	0.42	0	4.18	0	0.48
6	0	0.43	0	4.22	0	0.50
7	0	0.45	0	4.44	0	0.52
8	0	0.46	0	4.50	0	0.54
9	290	5.15	29	11.56	29	2.92
10	866	15.23	116	18.59	114	31.24
Prove Optimal	57,664	1,304.99	499	56.59	479	125.01

Table 3.2: Results of finding an optimal solution to an alien tiles problem using different symmetry breaking systems.

3.3 Implementing the GHK Algorithm with GAP

With the view of creating a symmetry breaking system to act as a library for Ilog Solver 5.2, we implemented a version of GHK-SBDS. The main differences between this implementation and that found in [GHK02] are firstly, the original version is written for ECL^iPS^e and this version is written for Solver 5.2. Secondly, the original version contained delayed goals to reason more intelligently with symmetries whereas this implementation does not. Finally, we implemented an additional layer of code that sits in between GAP and the constraint programmer that allows the symmetries to be described more easily. By doing so it is hoped to create a symmetry breaking system that has many of the desired properties mentioned in Chapter 3.1. More on how this is achieved can be found later in Chapter 3.5.3.

The initial hurdle in creating such a symmetry breaking system was the inter-process communication between the CSP Solver (Solver 5.2) and the CGT package (GAP 4.3). Whereas ECL^iPS^e contains predicates for starting new sub-processes, Solver does not. Code was written to allow C++ to start a GAP sub-process which Solver could communicate with during search by passing characters down pipes.

The method of symmetry breaking used is in the style of SBDS. After backtracking from a failed assignment, constraints are posted to rule out symmetrically equivalent assignments. A cut-down version of the pseudocode described in [GHK02] is used that does not require delayed goals. The constraints posted by this pseudo code break non-broken symmetries and many of these constraints are also unique. The only duplicate constraints that occur are as a result of a reordering of the assignments in the constraint e.g. $v_1 \neq 1 \vee v_2 \neq 2$ may be posted alongside $v_2 \neq 2 \vee v_1 \neq 1$. Note that although this greatly minimises the symmetry breaking constraints needed there can still be an exponential number (w.r.t. the arity of the constraint) of identical constraints. The algorithm for calculating the symmetry breaking constraints to be added is shown in Algorithm 3.3.1.

Algorithm 3.3.1: NUSBDSCONSTRAINTS$(H, prev, A)$

comment: H is the pointwise stabilizer of the current partial assignment

comment: prev is the list of constraints from the parent node

comment: A is the failed var=val assignment

$RT \leftarrow Orbit(H, A)$

while $prev.hasAnotherConstraint()$

do { **for each** $g \in RT$

do { $c \leftarrow (prev.getNextConstraint()$ || $\neg(g(A)))$

if $\neg c.isSatisfied()$

then $cons.add(c)$

comment: cons becomes prev for the child nodes

return $(cons)$

3.3.1 Analysis of Performance

As an initial test for the suitability of using Solver and GAP, the simpler U-SBDS symmetry breaking system.was implemented. In this case however, the group theory computation was performed by GAP unlike the implementation in Section 3.2 which used native C++ code. The run-times produced by using U-SBDS with Solver and GAP show that the system is not useful in practice. In general, the saving in run-time generated by symmetry breaking does not validate the increase in run-time of using the system in the first place.

	U-SBDS					Solver 5.2		
n	GAP	Solver	Runtime	Fails	Sols.	Runtime	Fails	Sols.
4	0.07	0.01	0.09	3	1	0	4	2
6	0.08	0.01	0.25	12	1	0.01	36	4
8	0.27	0.10	0.61	85	12	0.02	324	92
10	3.46	1.47	6.11	1,327	92	0.22	5,942	724
12	77.18	37.04	141.66	29,325	1787	5.59	131,902	14,200
14	2,413.72	1,203.70	4,406.90	828,489	45,752	147.51	3,832,624	365,596

Table 3.3: Results of using GAP with a U-SBDS implementation in Ilog Solver 5.2 to solve the n-queens problem. The table shows the cpu-time taken for GAP and Solver as well as the actual overall run-time of U-SBDS

The system was tested by trying to find all solutions of the n-queens problem. The preliminary results for this experiment alone showed that this implementation was not worth pursuing further. Table 3.3 shows the results of the GAP based implementation of the U-SBDS algorithm. The overall run-time using this symmetry breaking system is greater than using no symmetry breaking at all.

It is not known for certain why such a symmetry breaking system should exhibit such poor run-times. We conjecture that it could be to do with the message passing between the two processes and/or the difference in speed of GAP versus Solver.

To address the first argument in more depth, when GAP is given a fail point, it calculates the relevant symmetrically equivalent fail points and transmits to Solver the images of those symmetries. The size of this information transmitted by GAP is $\mathcal{O}(cons \times points)$ where $cons$ is the number of constraints to be posted and $points$ is the number of points that the permutation group acts on. The amount of data being passed could mean that a lot of the run-time is being spent on I/O on the pipes between Solver and GAP.

Secondly, if we compare the cpu-times of Solver and ECL^iPS^e with respect to solving the dodecahedron 3-colouring problem (Table 3.5) [GHK02], we can see that Solver takes less time, whereas the GAP cpu-times should in theory be similar. It is possible that the GAP cpu-times are a higher percentage of the total run-time and thus the effect of symmetry breaking is detrimental.

3.4 Implementing the GHK Algorithm without GAP

Assuming the above conjecture is true, that the large run-times of the symmetry breaking system are as a result of large message passing and relatively high GAP cpu-times, one way to get around such problems is to write native code to efficiently perform the necessary group theory code therefore eliminating the need for GAP and its inter-process communication. In order to do so, certain group theory algorithms need to be written in C++ so that they can be compiled along with Solver code. Fortunately, GHK-SBDS only utilises a few of the many GAP algorithms available.

3.4.1 Group Theory Implementation issues

The following group theoretic capabilities are needed in order to implement the GHK-SBDS algorithm:

1. A system of storing and describing a group via a set of generator permutations
2. A method of enumerating the elements of the group and calculating its size[4]
3. An efficient implementation of the Orbit-Stabilizer algorithm

The first two requirements are fairly trivial while the third is a specialised requirement. There is a standard Orbit-Stabilizer theorem [But91] and there is a standard Orbit-Stabilizer algorithm [Ser03] to calculate the set of orbits or the stabilising subgroup using Schrier vectors. The complexity of group theory as it is, there are many optimisations that can be made to the Orbit-Stabilizer algorithm for specific groups. However, this symmetry breaking system assumes two things that increase the efficiency drastically.

Firstly, this Orbit-Stabilizer algorithm can only work on individual points (i.e. assignments) and not tuples or sets of points (i.e. partial assignments). This assumption allows the algorithm to verify if a new point is unique or not in $\mathcal{O}(1)$. Secondly, basic implementations of the Orbit-Stabilizer algorithm generally return many redundant generators. It isn't a trivial process to discover which of these generators are redundant. The complexity of this

[4]This will be mainly used for verification purposes.

Orbit-Stabilizer algorithm is $\mathcal{O}(gen \times |orbit|)$ where gen is the number of generators. Thus, for future calls to the Orbit-Stabilizer algorithm, we want the groups used to have as few generators as possible. This is done by taking the first 10 generators. While this is efficient and it is highly unlikely that it will not recreate the entire group, it is not a certainty. If only a subgroup of a possible larger group is created by using this Orbit-Stabilizer algorithm, the symmetry breaking system may not break all symmetry and thus can't guarantee unique solutions.

3.5 NuSBDS

We now introduce a new symmetry breaking system: Nu-SBDS. It is based on the GHK-SBDS algorithm and uses native group theoretic code to calculate the relevant stabilising groups and orbits. This is previously completed research by Gent, Harvey and Kelsey. The actual contribution that this symmetry breaking system adds to the symmetry breaking research community is the way symmetries are described. As was stated at the start of the chapter, we need to make a symmetry breaking system that can be added to a constraint solver as a singular module in the same way arc consistency algorithms and different search strategies are. We then looked at the requirements of a good symmetry breaking system and noted that they must be easy to use for constraint programmers with little or no knowledge of symmetry breaking techniques or research. NuSBDS provides such an ease of use by allowing the constraint programmer to describe their symmetries via a set of *macros*. Since NuSBDS is an implementation, it is specific to both Ilog Solver and the GHK-SBDS algorithm. The main theme of this research can be used for different constraint solvers *and* different symmetry breaking algorithms.

The main ethos behind NuSBDS is that it should be as easy to use as possible. The constraint programmer does not need to supply extra code, or a group and a consistent labelling of the CSP. The symmetries of the CSP are given in generic terms which allows the constraint solver to break symmetry in a general CSP problem and not for specific instances. Also, since the constraint programmer describes the symmetries of their problem by using a set of macros, no great coding effort is expended.

Presented below is pseudo code for how such a system could be used to describe symmetries:

```
Vars nqueen = Vars[1..n]
nqueen.domain(1, 8)

... constraints ...

Symmetry s(nqueen, ASSIGN)
s.add(SQUARE)
search(nqueen)
```

In the above example, the constraint programmer describes what the symmetries act on, either variables, values or assignments. They then describe the symmetries of the problem and leave the symmetry breaking system to do two things. Firstly, the method of symmetry breaking is not specified by the constraint programmer, thus it has been removed from their view. This satisfies our wish for a modular approach to symmetry breaking in the same way we use consistency algorithms or search traversals. Secondly, the method of describing the symmetries and interfacing with the symmetry breaking method is also hidden from the constraint programmer. This greatly reduces the amount of work and expertise needed to perform symmetry breaking.

We now imagine another example CSP that has more than one type of symmetry, the most perfect magic squares problem (see Appendix C.6).

```
Vars most = Vars[1..n*n]
most.domain(1, n*n)

... constraints ...

Symmetry s(most, VAR)
s.add(SQUARE)
s.add(CYCLE_ROW)
s.add(CYCLE_COL)
search(most)
```

The example code for the most perfect magic squares problem contains two significant differences from the n-queens problem code. Firstly, we say that the symmetries act on

variables. Secondly, we describe more than one type of symmetry. Note however, that we are still removed from the details of the symmetry breaking and rely on the constraint solver to combine all the symmetries described and interface correctly with the symmetry breaking system.

3.5.1 NuSBDS code examples

This section shows actual code fragments from CSPs that use NuSBDS to break symmetry. As such the following sections of example code will be clearer to readers already familiar with C++ and more specifically Ilog's Solver constraint solving toolkit.

The macros used in NuSBDS are essentially functions representing types of symmetries. When called, they look at the model of the CSP, perform some basic error checking to see if the symmetry can be applied to the model, and internally store the generators for those symmetries. Different macros can be called repeatedly so that different types of symmetry can be combined to create direct products of groups. This allows the constraint programmer to describe complicated groups by using a few simple commands.

For example, say we have an encoding of the n-queens problem which has n variables, where the value represents which column the queen should be placed in. In this model, the symmetries of a square act on the assignments. Using Solver we require an `IloGoal` object to search on, which we create like so, where `x` is the array of variables and `env` is an `IloEnv` object. The important facts to take from this are that `x` contains a list of constrained integer variables and the `IloGoal` object will allow us to call functions that will search to find values for the variables in `x`.

```
IloGoal goal = IloGenerate(env, x);
```

We can use the following code to break the symmetries of the problem, where `solver` is an `IloSolver` object.

```
Symmetry* sym = new (env) Symmetry(env);
IloIntArray type(env, 1, SQUARE);
IloGoal goal;
```

```
if(symBreaking){
    goal = NuSBDSGenerate
        (env, x, sym->setup(x, solver, ASSIGN, type));
} else{
    goal = IloGenerate(env, x);
}
```

The above code contains some new terms that need explanation. Firstly, the `Symmetry` class is a part of NuSBDS. We use this to describe and break the symmetries of the CSP. The `type` object is simply an array of integers. The `#define` statement in C++ was used to associate each macro with an integer, and the `type` array merely contains the list of integers (representing macros) to use. The `NuSBDSGenerate` function takes an additional parameter to the `IloGenerate` function i.e. the `Symmetry` object we created to deal with the symmetry. Note that we must also call the function `setup`, which records which macros to use and whether or not the symmetry acts on assignments (`ASSIGN`) or variables (`VAR`).

As another example, say we have the BIBD problem (CSPLib problem: prob028 [GW99], Appendix C.2) which we can model as a matrix of 0/1 variables, given any solution we can permute the rows and columns to yield another. This matrix may not necessarily be a square so in order to describe the group, we need to tell NuSBDS the number of columns. We can then simply use two macros to describe the symmetries of this problem. Notice that these macros have "rectangle" in their name to show that they potentially act on a non-square matrix[5].

```
Symmetry* sym = new (env) Symmetry(env);
sym->setNumOfColumns(numOfCol);
IloIntArray type
    (env, 2, SYMMETRIC_RECTANGLE_ROW, SYMMETRIC_RECTANGLE_COL);
IloGoal goal;
if(symBreaking){
    goal = NuSBDSGenerate
        (env, x, sym->setup(x, solver, VAR, type));
} else{
    goal = IloGenerate(env, x);
```

[5]Though all square matrices are also rectangles, a macro for squares is provided for simplicity.

}

Given a problem that needs to use more macros, we just create a larger `type` array where each element represents the macro to be used. Currently there are 11 different macros which can be used for either symmetries acting on assignments (`ASSIGN`) or variables (`VAR`). Here is the list of macros for variable symmetry:

- SQUARE - n^2 variables with the symmetry of an $n \times n$ square acting on them
- CYCLE_ROW - n^2 variables make a square where the rows can be cycled
- CYCLE_COL - n^2 variables make a square where the columns can be cycled
- SYMMETRIC_ROW - n^2 variables make a square where the rows are interchangeable
- SYMMETRIC_COL - n^2 variables make a square where the columns are interchangeable
- SYMMETRIC_RECTANGLE_ROW - as SYMMETRIC_ROW but for non-square matrices
- SYMMETRIC_RECTANGLE_COL - as SYMMETRIC_COL but for non-square matrices

Here is the list of macros for variable and value symmetry:

- SQUARE - e.g. n-queens
- SYMMETRIC_VAR - interchangeable variables
- SYMMETRIC_VAL - interchangeable values
- SQUARE_VAR - n^2 variables with the symmetry of an $n \times n$ square acting on them

The reader should note that some classes of symmetry can be broken more easily than using the underlying GHK-SBDS algorithm e.g. freely interchangeable values (also known as indistinguishable values), as found in graph colouring, can be broken using the constraint

described in [Gen01], using the technique in [HFPA03] or as is noted in [BW99], only a subset of the constraints posted by SBDS is needed to break all these symmetries. NuSBDS is most useful for combinations of symmetries that result in groups for which there are no efficient methods of dealing with.

3.5.2 Comparison of symmetry description methods

In order to demonstrate the ease of use of NuSBDS further, we will present example code that **describes** symmetries for other symmetry breaking systems. By doing so, we intend to make a convincing argument that by using the macros in NuSBDS, symmetries can be described more easily and naturally than with any other symmetry breaking method. A common example CSP used throughout this thesis is the n-queens problems. We now give code for implementations of different symmetry breaking systems.

NuSBDS

Below is the bare minimum necessary to describe the symmetries of the n-queens problem with NuSBDS. It consists of naming the one symmetry macro for the problem (i.e. the symmetry of a square) and associating it with symmetries acting on assignments. Note again the main advantages of this system, the concise representation (just 3 lines) and the natural language description.

```
Symmetry* sym = new (env) Symmetry(env);
IloIntArray type(env, 1, SQUARE);
IloGoal goal = NuSBDSGenerate
    (env, x, sym->setup(x, solver, ASSIGN, type));
```

SBDS

The following code sample is taken directly from the paper [GS00]. They represent the symmetries of the n-queens problem. SBDS requires an explicit list of the symmetries of the problem. In this case, there are 7 symmetries (not including the identity) but for more symmetric problems the number of functions would be greatly increased. A function of

this sort would need to be both produced (either by hand or with a scripting language) and compiled (in this implementation by a C++ compiler).

Thus it is clear that for symmetric problems, even if SBDS can handle the number of symmetries required, great effort is extended in producing the symmetry functions.

```
IlcConstraint r90 (IlcIntVarArray vars, IlcInt i, IlcInt j)
    {return vars[j] == n-1-i;}
IlcConstraint r180 (IlcIntVarArray vars, IlcInt i, IlcInt j)
    {return vars[n-1-i] == n-1-j;}
IlcConstraint r270 (IlcIntVarArray vars, IlcInt i, IlcInt j)
    {return vars[n-1-j] == i;}
IlcConstraint x (IlcIntVarArray vars, IlcInt i, IlcInt j)
    {return vars[n-1-i] == j;}
IlcConstraint y (IlcIntVarArray vars, IlcInt i, IlcInt j)
    {return vars[i] == n-1-j;}
IlcConstraint d1 (IlcIntVarArray vars, IlcInt i, IlcInt j)
    {return vars[j] == i;}
IlcConstraint d2 (IlcIntVarArray vars, IlcInt i, IlcInt j)
    {return vars[n-1-j] == n-1-i;}
```

SBDD

The following code was used for the empirical experiments performed in [FSS01][6]. The symmetry description in this case is very similar to that of SBDS above. One main difference is that SBDS needs separate functions for each symmetry but each symmetry is represented in SBDD as an condition in a switch statement. Since the n-queens problem has just 7 symmetries (not including the identity), all symmetries are described explicitly. For exponentially symmetric CSPs, the generator set of symmetries can be described and SBDD will detect dominance under the product of these symmetries.

Thus, SBDD has an advantage over SBDS in that drastically fewer symmetries need to be described. However, the process of describing these necessary symmetries is no more straightforward.

[6]Thanks to Meinolf Sellmann for allowing his code to be included in this thesis.

```
switch(k)
{
    case 0: // left-right *
        for (l=0; l<nonZero; l++)
        {
            i=list[l];
            h[n-1-i]=pattern[i];
        }
        break;
    case 1: // up-down *
        for (l=0; l<nonZero; l++)
        {
            i=list[l];
            h[i]=n-1-pattern[i];
        }
        break;
    case 2: // 180 *
        for (l=0; l<nonZero; l++)
        {
            i=list[l];
            h[n-1-i]=n-1-pattern[i];
        }
        break;
    case 3: // 90 *
        for (l=0; l<nonZero; l++)
        {
            i=list[l];
            h[pattern[i]]=n-1-i;
        }
        break;
    case 4: // 270 *
        for (l=0; l<nonZero; l++)
        {
            i=list[l];
```

```
            h[n-1-pattern[i]]=i;
        }
        break;
    case 5: // d2
        for (l=0; l<nonZero; l++)
        {
            i=list[l];
            h[n-1-pattern[i]]=n-1-i;
        }
        break;
    case 6: // d1 *
        for (l=0; l<nonZero; l++)
        {
            i=list[l];
            h[pattern[i]]=i;
        }
    break;
}
```

GHK-SBDS and GHKL-SBDD

Although these two methods of breaking symmetries differ greatly, the method of describing symmetries is almost identical since they both require a group to be passed to a GAP subprocess of the constraint solver. This group can be expressed explicitly as a generator set of permutations, or implicitly as a GAP program that can be used to produce problem specific symmetries.

The following code shows instance specific code for creating a permutation group to model the symmetries of the 3-queens problem.

```
gap> g := Group((1,3,9,7)(2,6,8,4), (1,3)(4,6)(7,9));
```

The constraint programmer also needs to produce code to map the points to assignments and vice versa e.g. point 4 is equivalent to $X_2 = 1$.

By describing symmetries as groups we retain the benefits of concise representation as NuSBDS does. However, background knowledge of group theory is needed to use GHK-SBDS and/or GHKL-SBDD. The system of natural language keywords that NuSBDS has, alleviates the need for this.

Conclusions

Though the methods of breaking symmetries in the above examples are totally different, the task of describing symmetries in these previous symmetry breaking systems is very similar. They are all concerned with describing how a symmetry applies to the constraint model being used. The other main similarity with all the symmetry breaking systems is that the constraint programmer must produce either code, functions or an object/attribute that the system then directly uses to break symmetry.

NuSBDS places a layer of abstraction between the constraint solver and the symmetry breaking system. This allows the constraint programmer to create the input required by the symmetry breaking system indirectly. As the above examples illustrate, this indirect process makes describing symmetries simpler for the constraint programmer.

3.5.3 Macros

The macros system implemented in NuSBDS is the unique way that symmetries are described. Each macro is an abstraction layer that takes some user defined parameter (in most cases the number of constrained variables) and outputs a group acting on the CSP. Rather than return this output to the user, it is stored internally by NuSBDS to be used to break symmetry. The constraint programmer never sees the group itself. NuSBDS can also take the groups created by the various macros and combine them to create a larger group. Again, this group is hidden from the constraint programmer and used to break symmetry.

In order to combine macros to generate larger groups, a consistent action set must first be chosen by the constraint programmer. If the symmetries of a certain problem act only on the variables, the constraint programmer can say that the action set will be the variables or the assignments. If some symmetries act on the variables and some on the values, the constraint programmer must say the symmetries act on the assignments. Once the action set has been

decided, the groups returned by different macros will all be on the same points. It is the responsibility of the constraint programmer to ensure that the macros are used correctly to produce groups that represent symmetries that satisfy Definition 1.7 for the given CSP.

Consider the groups H, J and G such that $H \times J = G$. We can recreate all the elements of G by finding the *direct* product of the elements of H and J. If H and J are both permutation groups that **act on the same points**, we can find generators for the group G by taking the union of the generator sets of H and J. This is indeed how NuSBDS combines the groups returned from the various macros called.

Research by Harvey et al. [HKP03] can create groups by taking the direct product of two groups acting on the same number of points as NuSBDS does. They can also create groups by taking the wreath product of two groups as well, something that NuSBDS cannot do. Consider a group K, acting on 30 points and a group L acting on 3 points. We can combine these groups by abstracting the points L acts on e.g. treat the first 10 points K acts on as point 1, the next 10 as point 2 and the final 10 as point 3. By combining groups in such a way we have created the *wreath* product of K and L. This functionality is useful for common models of the golfers' problem [HKP03].

Macro Implementation

Each macro is represented as a word such as `SQUARE`, `SYMMETRIC_COL` etc. The `#define` command has been used to link every word with an integer:

```
#define SQUARE 0
#define SQUARE_VAR 1
#define SYMMETRIC 2
...
```

The constraint programmer selects the relevant macros and this information is used to call the correct function. Before calling the macro function, some basic error checking is performed if possible to see if the symmetry is valid for the given CSP e.g. if the symmetry is that of a square acting on the assignments of the CSP, the size of each of the domains must be the same as the number of variables. Also, NuSBDS insists that all domains must be consecutively numbered starting from 0. Then the macro function is called with the

relevant parameter based on the size of the CSP. By doing so, we can deal with symmetry for a specific type of CSP, not just a specific sized instance of a CSP.

Here is an example of the code one of the macro functions. Each macro featured in NuS-BDS takes the form of a method like this. This particular macro returns the generators of the group representing the symmetries of a square acting on the assignments of a CSP with n variables. Note that the group representing the symmetries of a square needs only two generators: flip around an axis and rotate 90° (stored in `p[0]` and `p[1]` respectively). Each cycle in the permutation is stored as an `IlcIntArray` object, which is an array of integers.

```
Permutation* Symmetry::perm_square(IloInt n){
    Permutation* p = new (e) Permutation[2];
    p[0].num = n * (n/2);
    p[0].g = new (e) IlcIntArray[p[0].num];
    IloInt next = 0;

    for(IloInt j=0; j < n; j++){
        for(IloInt i=1; i <= n/2; i++){
            IlcIntArray temp(sol, 2, j*n + i, j*n + n-i+1);
            p[0].g[next] = temp;
            next++;
        }
    }
    next = 0;
    for(IloInt i=0; i < n/2; i++){
        for(IloInt j=1; j <= n-2*(i+1)+1; j++){
            next++;
        }
    }
    p[1].num = next;
    p[1].g = new (e) IlcIntArray[next];
    next = 0;
    for(IloInt j=0; j < n/2; j++){
        for(IloInt i=1; i <= n-2*(j+1)+1; i++){
            IlcIntArray temp(sol, 4, i+j*(n+1), i*n+n*j-j,
```

```
                n*n-i-n*j-j+1, n*n-i*n+1-j*(n-1));
            p[1].g[next] = temp;
            next++;
        }
    }
    return p;
}
```

More detailed examples and specifics about NuSBDS can be found in Appendix B - the NuSBDS user manual.

3.5.4 Empirical Comparisons

In this section, the empirical performance of NuSBDS is tested on some symmetric CSPs (See Table 3.4, Table 3.5, Table 3.6, Table 3.7 and Table 3.8). NuSBDS is based on the GHK-SBDS implementation and as such, can handle over 10^7 symmetries. However, even now this number is not as impressive as results matched by other symmetry breaking techniques [GHKL03] [Pug02]. The main feature of NuSBDS though is user friendliness. We present empirical data merely to show that such a symmetry breaking system can be used to break symmetries efficiently. A superior symmetry breaking system could be implemented and still have the ease of use of NuSBDS so long as the system and the macros interfaced with each other correctly.

The size of the list of constraints to be posted by GHK-SBDS increases exponentially with the depth of the search tree. This will eventually use all the memory available to the computer and the solver will spend more time on memory (de)allocation than solving. NuSBDS overcomes this problem by setting a bound[7] on the maximum number of symmetry breaking constraints that are allowed to be stored at any one time. While this reduces the amount of symmetry breaking, the solver retains a polynomial bound on memory requirements. Removing the exponential bound on memory requirements, the constraint programmer is free to describe as many symmetries as possible. In examples with highly symmetric problems though e.g. BIBD, the number of solutions reported by NuSBDS will most likely not be the number of unique solutions. A table of results of finding solutions to BIBDs using

[7] By default of the order of 10^5, but can be specified at runtime.

	Solver 5.2			NuSBDS		
n	Sols.	Runtime	Fails	Sols.	Runtime	Fails
4	2	0.00	4	1	0.00	3
6	4	0.00	35	1	0.00	11
8	92	0.02	289	12	0.00	61
10	724	0.22	5,072	92	0.09	875
12	14,200	4.78	103,956	1787	1.52	17,801
14	365,596	137.92	2,932,626	45,752	47.83	485,128

Table 3.4: Results of using NuSBDS and Solver 5.2 to find all solutions to various n-queens problems.

	Solver 5.2			NuSBDS		
Dodecahedron	Sols.	Runtime	Fails	Sols.	Runtime	Fails
3-colouring	7,200	0.09	132	31	0.04	19

Table 3.5: Results of using NuSBDS and Solver 5.2 to find all 3-colourings of the dodecahedron. This problem has 360 symmetries. They consist of the 3! or 6 symmetries from the 3 available colours combined with the 60 symmetries of the dodecahedron itself.

constraint programming can be found in [Pug03]. A more complete reference can be found in [MR90].

		Solver 5.2			NuSBDS		
n	Syms.	Sols.	Runtime	Fails	Sols.	Runtime	Fails
4	128	384	0.48	1,594	3	0.07	73
6	288	0	2,905.12	4,176,447	0	14.85	25,157

Table 3.6: Results of using NuSBDS and Solver 5.2 to solve the most perfect magic squares problem.

Alien Tiles (4, 3)	Runtime	Fails
Solver 5.2	1004.58	54,081
NuSBDS	15.55	482
SBDS	21.72	493

Table 3.7: Results of using NuSBDS, SBDS and Solver 5.2 to solve the alien tiles problems for a 4×4 board with 3 colours.

BIBD		Solver 5.2			NuSBDS		
v b r k λ	Syms	Sols.	Runtime	Fails	Sols.	Runtime	Fails
4 6 3 2 1	17,280	720	0.02	29	1	0.08	11
7 7 3 3 1	2.5×10^7	151,200	8.93	11,680	92	0.53	42
7 7 4 4 2	2.5×10^7	151,200	10.91	64,639	234	7.51	43
5 10 4 2 1	4.3×10^8	3,628,800	109.59	113,291	5,400	0.87	1,233

Table 3.8: Results of using NuSBDS and Solver 5.2 to solve some small BIBDs. Finding all solutions to the BIBD problem is generally hard due to the number of symmetries. NuSBDS does not break all symmetry which allows a more efficient symmetry breaking system.

3.6 Conclusions and Future Work

In this chapter we have looked at various different symmetry breaking systems. We have considered the different functionality of these systems. We have proposed standards that implementations of these symmetry breaking systems should aim toward in order to be easily integrated into constraint solvers.

We introduced the first implementation for breaking general user specified symmetries in constraint programming with group theory. By introducing a group theoretic symmetry description we eliminate the disadvantage of listing all symmetries. For previous symmetry breaking systems such as SBDS, this disadvantage greatly reduced the size of groups that could be handled. All modern general purpose symmetry breaking systems [FSS01] [GHK02] [GHKL03] [Pug03] now use group theory ideas e.g. a generator set of symmetries, to break very large groups of symmetry. We also showed that by using group theoretic concepts, SBDS and similar symmetry breaking systems can break all symmetry by posting a very small subset of all possible symmetry breaking constraints. We presented a general upper bound for the maximum number of constraints needed upon backtracking from any failed assignment. We also reasoned that this number would be much smaller in practice when accounting for the intersection with non broken symmetries as well.

By concentrating on the interface between the process of describing symmetries and the symmetry breaking system, we developed a truly user friendly method of describing symmetries. This idea led to the creation of NuSBDS, a symmetry breaking system based on the GHK-SBDS [GHK02] algorithm. NuSBDS uses a series of macros that can be safely combined to describe many different groups. These macros allow the constraint programmer to easily describe symmetries and more importantly allow a general CSP to be scaled to produce different sized instances. Thus, once the symmetries have been described for a given CSP, they are automatically scaled at runtime to all instances.

We objectively examine the advantages and disadvantages of general purpose symmetry breaking systems. We highlight desirable aspects of various systems, and introduce aspects which are fundamental to the inclusion of symmetry breaking systems in future constraint solvers. There is still much work to do however before we can see truly industry standard symmetry breaking systems.

3.6.1 Improved Symmetry Breaking Techniques

There are now symmetry breaking techniques that have surpassed the performance of the GHK-SBDS algorithm included in NuSBDS. The SBDD [FSS01] and GHKL-SBDD [GHKL03] algorithms can be used to break general user specified groups of much larger size. Also, though STAB [Pug03] has currently only been used to break symmetry in matrix models, more intelligent implementations should be able to break user described symmetries.

The ideas in this chapter can easily be extended so that the symmetry breaking systems mentioned above (and other new ones) can make use of macros to succinctly and easily describe symmetries. Such a combination of superior symmetry breaking and easy symmetry description would be a most welcome addition to any constraint solver.

3.6.2 User specified macros

The dodecahedron colouring problem (see Figure 3.5) was specifically chosen as a CSP that exhibited an uncommon group. Thus, the symmetries were not described via any of the NuSBDS macros but by a specific group. Though being able to describe symmetries via group generators would be a welcome addition to a symmetry breaking system, a better solution would be to allow the constraint programmer to construct their own macros. This way, the symmetry breaking system would grow to be able to solve complex symmetrical problems.

3.6.3 Intelligent Symmetry Breaking

In general, there is no tractable method for breaking any group of symmetries. There are however, some types of symmetry that can be broken easily. Lex ordering an array of symmetrically equivalent variables breaks all symmetry. Efficient methods for breaking interchangeable value (or indistinguishable value) symmetry have been developed by Gent [Gen01] and van Hentenryck et al. [HFPA03]. If the symmetries of the CSP deal with just one of these types of symmetry, then a general method of symmetry breaking such as GHK-SBDS may not be preferable.

The macros used to describe the symmetries of the CSP could be provided with information to automatically select the most appropriate symmetry breaking system. Again, this would take the complexities of various symmetry breaking systems away from the constraint programmer.

3.6.4 Partial Symmetry Breaking

Partial Symmetry Breaking is the focus of Chapter 4. NuSBDS has some simple partial symmetry breaking which limits the maximum number of constraints that can be added to the constraint solver. This small optimisation stops NuSBDS from using all the memory resources.

The research carried out in the partial symmetry breaking chapter (which partly appears in [MS02]) describes methods for choosing good symmetries to break. Implementations of symmetry breaking systems that are aware of such methods will be able to break more symmetry and thus solve problems with less computation.

3.6.5 Verification of Symmetries

NuSBDS contains some basic checks to try to ensure that the constraint programmer selects valid macros for their CSP. However, the consequences of incorrectly describing symmetries is disastrous. Valid solutions will most likely be rejected and in extreme cases, no solutions may be reported.

Graph automorphism checks could be implemented that could verify the symmetries described using macros. The advantage of parameterised CSPs could be used so that a check could be performed on a small instance of a symmetric CSP which would probably hold for larger instances. If such a procedure were to be implemented, care must be taken to safely deal with symmetries that exist that the graph automorphism check does not detect.

Of all the possible directions for the future work of general purpose, easy to use symmetry breaking systems, the participation of constraint programmers is the most important. Constraint programmers not familiar with sophisticated methods of breaking symmetry will be able to show constraint solver developers how best to include symmetry breaking systems.

Chapter 4

Partial Symmetry Breaking

In this chapter we define *partial symmetry breaking*, a concept that has been used in many previous papers without being the main topic of any research. This chapter is the first systematic study of partial symmetry breaking in constraint programming.

4.1 Introduction and Motivation

Given a CSP with a set of constraints and variables, it is possible to use propagation techniques to infer solutions i.e. for n variables, enforcing n-consistency allows us to find all solutions to the problem. In practice however this is not a viable method of solving CSPs as it uses an exponential amount of memory. We resolve the problem of expensive consistency requirements by limiting ourselves in general to performing some local consistency.

In a similar manner it is theoretically possible to break all symmetry for any given group. However this is often not achievable in practice. There are methods of breaking very large numbers of symmetries e.g. lexicographically ordering n objects breaks the $n!$ symmetries of S_n. Focacci and Milano also describe a filtering algorithm for breaking $n!$ symmetries that runs in $O(nd)$ where d is the size of the largest domain [FM01]. As was mentioned in Chapter 2.4, the former method can behave badly if used with incompatible heuristics and the latter can only be used for a specific "family" of symmetries with the nogood recording method of symmetry breaking. Also, the symmetric group[1] acting on variables or values is

[1]The symmetric group of size n has $n!$ elements.

significantly easier to break than other groups with wreath or direct products. For groups of symmetry in general, there is no tractable method of breaking an exponential number of symmetries. For the case where we have too many symmetries to be able to handle them efficiently, it is necessary to break a subset of all possible symmetries.

If we consider a symmetry breaking system, there are two distinct parts. Firstly there is the symmetry breaking *technique*. This is the algorithm or function that takes a symmetry and performs some task ensuring that that symmetry is broken. For example, SBDD [FSS01] looks for a symmetry and compares the mapping of that symmetry against all the book-marked nogood nodes with the current node. If any of the symmetric states are a subset of the current state then we should backtrack. The SBDS algorithm [GS00] will take a symmetry and if it is not guaranteed to be broken already, a symmetry breaking constraint is added to the local subtree. The cost of performing either of these algorithms to break one symmetry is trivial. The reason we have to limit the amount of symmetry breaking it is possible to do is that these steps mentioned above need an exponential amount of run-time to complete. Even though it's possible to reduce the number of symmetries to consider, if there is an exponential number of symmetries, the run-time of performing symmetry breaking itself becomes exponential. Symmetry breaking systems that add constraints to break symmetry can also add an exponential number of constraints. This in turn has a devastating effect on propagation algorithms that have a time complexity proportional to the number of constraints e.g. arc consistency.

Secondly there is the symmetry *representation*. This is the method of storing a given symmetry (or set of symmetries) that are to be used by a given symmetry breaking technique. Every time the symmetry breaking procedure is called upon, the technique breaks all the symmetries that are contained within the symmetry representation. Therefore, if there are an exponential number of elements in the symmetry representation, the symmetry breaking will have an exponential runtime.

We can now see that it is not the symmetry breaking *technique* that is the overhead in performing symmetry breaking but rather it is the number of symmetries to break in the symmetry *representation* associated with the system. We balance the propagation used when solving CSPs with the search performed in order to minimise run-time. In a similar way, we must balance the benefit of applying symmetry breaking to avoid redundant search with the expense of the symmetry breaking system itself. While much research has observed that breaking a subset of all symmetries is a valid and sometimes necessary pro-

cedure, this chapter is the first systematic study of *partial symmetry breaking* in constraint programming.

4.2 Review of Partial Symmetry Breaking

As previously mentioned , other papers have used partial symmetry breaking (henceforth refered to as PSB) on problems with large amounts of symmetry. There now follows a review of the brief experiments involving partial symmetry breaking and an analysis of how best to perform partial symmetry breaking.

In the original SBDS paper by Backofen and Will [BW98], SBDS is used to break symmetry on the photo problem, which is a problem with freely interchangeable symmetry on the values. They use SBDS to break just the transposition symmetries since there are a polynomial number of them, and they report that this subset breaks all symmetry. Their thinking behind this is also that the transpositions being a *generating set* for S_n might also be relevant.

They do claim though that, in general, breaking a generating set does not break all symmetry. The paper states that the reason this is the case for S_n on values is "an open question." They go on to prove that the transpositions break all symmetry in their next paper [BW99]. Since then, there have been other efficient symmetry techniques that break freely interchangeable value (or indistinguishable value) symmetry by Gent [Gen01], van Hentenryck et al. [HFPA03] and Gent et al. [RDGKL04].

In [GHK02], Gent and Smith include a section titled "A Restricted SBDS Method" where they describe how to perform PSB with SBDS. If there are too many symmetries to deal with, it is perfectly valid for a constraint programmer to list a subset of all of them. This will return valid solutions however, it no longer guarantees unique solutions. As an example they state that, "a graph k-colouring problem has $k!$ symmetries. Direct use of SBDS is then impractical." They suggest using just the symmetries for which the pre-condition for the symmetry breaking constraints (i.e. $g(A)$) is guaranteed true. In group theory terms, this means using the symmetries in the stabilizer of current state A.

Smith uses PSB in [Smi01] to make the golfers' problem more tractable. When limiting the symmetries to be broken by SBDS, it is claimed, "It seems intuitively plausible that the

simpler symmetries might give highest returns, as well as being easiest to describe."

The original SBDD paper by Fahle, Schamberger and Sellmann [FSS01], describes a dominance check that contains a subset of all symmetries when solving instances of the golfers' problem. They reason, "Since invoking the symmetry detection function ... is computationally very expensive, applying it in every search node does not improve the overall runtime, although the number of choice points is reduced. Thus, there is a trade-off between the reduction of choice points and the effort spent for the detection of symmetries." They devise a test for breaking all symmetries at leaf nodes (to ensure unique solutions) and every q^{th} level. For the golfers' problem with 4 groups of 4 players for 4 weeks, they conclude that "an invocation in about every 8^{th} level has shown to be the best."

Puget's CP2002 paper [Pug02], states that, "It is often better not to remove all symmetries ... Moreover, although this is not displayed here because of lack of space, the time needed to get the first solution is also greatly improved in such case, and is comparable to the time without symmetry removal." In Puget's paper the following year [Pug03], there is more detail about how the PSB was performed. The STAB technique is used to only 70% of the depth of the search tree which yields on average (according to the experiments in the paper) a 17% improvement in runtime. The STAB technique itself however, is already using PSB since it does not guarantee to break all symmetry (See Chapter 4.6).

Aloul et al. [ARMS03] break just an *irredundant generator set* of symmetries. They state that "one can often achieve significant pruning because an irredundant set of generators contains 'maximally independent' symmetries i.e. none of them can be expressed in terms of others." A brief example of how different generator sets rule out different symmetrical solutions is presented. The authors do not give scientific reasons as to why the ideal symmetries to break must be a generator set or even an irredundant generator set. They claim that their future research will attempt to explain why some irredundant generator sets are preferable to others.

The above cases effectively demonstrate that PSB is a well known and highly rated technique for improving the runtime of symmetry breaking systems. Some of the research detailed here has shown that certain subsets of symmetry can break larger numbers of symmetries. Though some minor experiments have taken place to try and maximise a symmetry breaking systems performance, in many cases, little explanation is given and no general understanding is reached as to why using less symmetries results in greater runtimes.

It is the aim of this research to uncover why breaking less symmetries can reduce the runtime of constraint solving, and in doing so, present constraint programmers with the information they need to maximise PSB in future experiments.

4.3 Definitions and Notation

The set of all symmetries of a CSP form a **group**. The way in which partial symmetry breaking is performed depends on the symmetry representation. Previous encodings have used group theory techniques to represent a large number of symmetries by listing a small subset of all of them [GHK02] [McD01] [BFP96] [GHKL03]. If it is possible to recreate the entire group of symmetries by reapplying the symmetries in this small subset, we call the subset a *generator set*. Many of the experiments and findings in this chapter are based on representations that encode all the symmetries of a CSP and not just the generator set. We discuss later how results from this chapter may be used with generator set representations.

We now define two classes of symmetric CSPs.

Definition 4.1 *Given a CSP L where the number of symmetries of L increases polynomially with respect to the sizes of the variables X and their domains $D(X)$, L is said to be* polynomially symmetric.

In the n-queens problem for example, the number of symmetries is 8 regardless of n. The most perfect magic squares problem (See Appendix C.6) has $8n^2$ symmetries for an $n \times n$ board. For these types of problems, SBDS by Gent and Smith [GS00] or Symmetry Excluding Trees by Backofen and Will [BW99] are probably the best methods of removing symmetry as the overhead is low. Though SBDD is a better approach for more symmetric problems, small problems like the n-queens are solved in less time and with fewer backtracks with SBDS [FSS01].

Definition 4.2 *Given a CSP L where the number of symmetries of L increases exponentially with respect to the sizes of the variables X and their domains $D(X)$, L is said to be* exponentially symmetric.

Naïve encodings of the exponentially symmetric golfers' problem (See Appendix C.5) have

$(\frac{p}{g}!)^{gw}(g!)^{w}w!p!$ symmetries for p players, g groups and w weeks [Har01]. Clearly, increasing either the number of players, groups or weeks by even one will greatly increase the number of symmetries.

4.4 Partial Symmetry Breaking and Symmetry Representation

There have already been two improvements reported on the representation of symmetries i.e. methods for removing symmetries from consideration from the symmetry representation. The first is found in the symmetry breaking method SET (symmetry excluding trees) developed by Backofen and Will [BW99] and this removes broken symmetries. Removing broken symmetries from consideration is also in the symmetry breaking method that will be used in the experiments in this chapter: SBDS. Therefore, the concept will be explained in terms of the SBDS notation. Symmetry Breaking During Search (SBDS), developed by Gent and Smith [GS00], works by adding constraints to the current search subtree. After backtracking from a failed assignment $var_i = val_j$, to a point in search with a partial assignment A, we post the constraint:

$$g(A) \ \& \ (var_i \neq val_j) \Rightarrow g(var_i \neq val_j)$$

for every g in the symmetry representation. Symmetries are represented by functions and SBDS removes a function from consideration when it discovers that a pre-condition (i.e. $g(A)$) of the constraint it creates is guaranteed false from the current subtree. For example consider a node k in search. A symmetry function may produce a pre-condition $var_i = val_j$ but if at point k, $var_i \neq val_j$ we can ignore that symmetry function at all child nodes of k.

The second improvement is found in Chapter 3.2 [McD01] where only unique symmetries are considered. We showed how at certain points in search, some sets of symmetries all have the same effect on a partial assignment hence we can discard all but one symmetry from this set. For example, there may be a pair of symmetries g and h such that given a partial assignment A, $g(A) = h(A)$. If this is the case we only break symmetry on g or on h but not on both. These two improvements reduce the number of symmetries to consider without reducing the amount of symmetry breaking possible i.e. they do not introduce

non-unique solutions.

We now consider a third optimisation. In this chapter we show that where there is a large number of symmetries, we can discard some of them and, by doing so, reduce run-time greatly. If we are trying to solve a problem that is exponentially symmetric we may not be able to fully utilise a given symmetry breaking technique. We cannot apply the dominance check used in SBDD for all symmetries at every node in search for the golfers' problem as it is too expensive [FSS01]. It is still possible to use SBDS if we limit the number of symmetry breaking functions, and we can still use SBDD by applying a dominance check under a subgroup of all the symmetries. Describing only a subset of the symmetries does not lose any solutions (and may result in redundant search) but the overhead of performing symmetry breaking will not be as great. By describing only a subset of symmetries we are performing PSB i.e. performing some redundant search because the symmetry breaking technique is too costly or even impossible to perform.

4.4.1 Explicit Symmetries and Group Theory

Given a symmetry representation, we perform PSB by only applying the symmetry breaking technique to a subset of the symmetries in the symmetry representation. How this subset of symmetries is generated depends on the symmetry representation. There are two types of symmetry representation:

1. A list of explicit symmetries
2. A generator set of a group

Generating a subset of symmetries from a list of explicit symmetries is trivial, however, the implicit nature of using generators of groups makes it difficult (but still possible) to select a subset of symmetries. This will be discussed in more detail later.

4.5 Partial Symmetry Breaking Experiments

It is a straightforward assumption that by breaking more symmetries i.e. by increasing the number of symmetries in the representation, we can reduce the search space further up to a

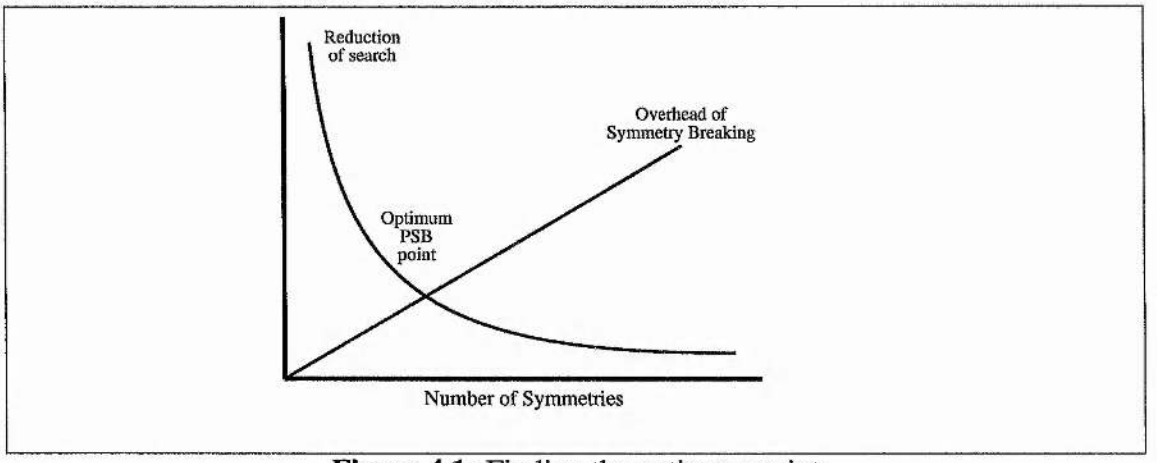


Figure 4.1: Finding the optimum point

certain point.

However, as the number of symmetries represented increases, so does the overhead. We will show how there is an optimum point of symmetry breaking as illustrated in Figure 4.1 by suggesting that there may be a point where the benefit in reducing search from adding more symmetries is out-weighed by the extra overhead.

In order to discover how the cpu-time of solving a symmetric CSP varies with the number of symmetries used in the symmetry representation we have constructed the following experiment. **We record the cpu-time of solving a CSP L, with a subset of the symmetries G, of L.** This is done for subsets of size 0 to $|G|$. If we require a subset H, of size h, of the set of symmetries G, of size g, there are gCh different subsets that H can be. Any time we select a subset of symmetries for a given experiment, we choose a pseudo-random subset. If it is necessary to repeat the experiment with different subsets we choose new seeds to generate other pseudo-random subsets.

Take a CSP L with n symmetries solved using a set of symmetry breaking functions k, with SBDS where $|k| \leq n$. For $|k| = 0$ to $|k| = n$ we find the cpu-time taken to solve L and use this information to plot points on a graph. The set of symmetry breaking functions used are chosen pseudo-randomly.

Given the data from the above experiment we can plot cpu-time against number of symmetries used. We can then use this graph to estimate how many symmetries we need to break to minimise cpu-time for SBDS. It should be highlighted though that by doing this we allow duplicate solutions. Unique solutions can be found by applying an SBDD dominance

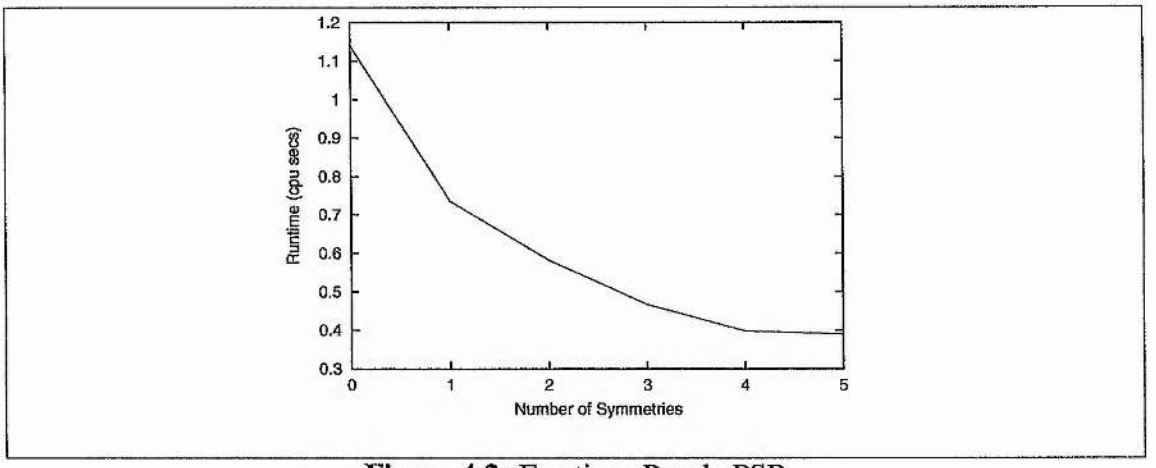


Figure 4.2: Fractions Puzzle PSB

check to leaf nodes [FSS01] or by some other means of isomorph rejection.

4.5.1 Fractions Puzzle

We consider a very simple problem as an example experiment (See Appendix C.4). Given the following problem:

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1$$

Can we find unique values (from the range $\{1..9\}$) for each variable such that the equation[2] is satisfied? Note that we can permute the fractions freely, yielding 5 symmetries and the identity e.g. one symmetry is $A \leftrightarrow D$, $B \leftrightarrow E$ and $C \leftrightarrow F$. Since the number of symmetries is so small it is possible to run the experiment with all possible subsets of symmetries. The cpu-times were then averaged for each subset size. Figure 4.2 contains the graph of the averaged cpu-time with respect to the number of symmetries. As you can see, by adding more symmetries the cpu-time decreases.

[2] BC does not mean $B \times C$ but rather $(10 \times B) + C$.

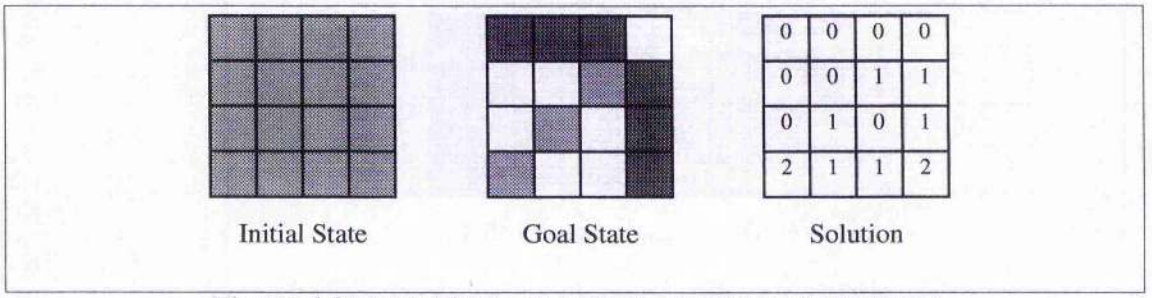


Figure 4.3: Initial State, Goal State and Example Solution

4.5.2 Alien Tiles

SBDS has already been used to solve alien tiles problems (See Appendix C.1) with good results [GLS00]. The alien tiles board can be described with two parameters n and c, the size of the board and the number of colours respectively. An alien tiles board is an $n \times n$ grid of n^2 coloured squares[3]. By clicking on any square on the board, the colour of the square is changed $+1$ *mod* c. As well as this, the colour of every square in the same row and column is also altered $+1$ *mod* c. Given an initial state and a goal state, the problem is to find the required number of clicks on each square which can be anything between 0 and $c-1$ (since $0 \equiv c$, $1 \equiv c+1$ etc). A more challenging problem for constraint programming (which can be found in CSPLib [GW99] - problem 27) is finding the most complicated goal state (in terms of the number of clicks needed) for some initial state and then reaching that goal state in as few clicks as possible and verifying optimality.

The problem we consider is a 4×4 board with 3 colours. Figure 4.3 shows the initial state and an optimally hard goal state for the problem we are trying to solve and an example solution. The smallest number of clicks that can take us to the goal state is 10. Proving that 10 clicks is optimal needs a complete traversal of the entire search tree.

An instance of the alien tiles problem is exponentially symmetric. Given a solution we can freely permute the rows and columns and flip the board around a diagonal. For a board with n^2 variables, the group acting on the board is $S_n \times S_n \times 2$ which for a 4×4 board is a group of size 1152, or 1151 symmetries and the identity. We derive this number by noting that we have 24 (or $4!$) row permutations, which can be used in conjunction with the 24 column permutations, which can be used with the diagonal flip ($2n!^2$). The reason we are using this symmetric CSP as the main example of PSB is that it is not a trivially easy

[3] Alien tiles puzzles can be found online at http://www.alientiles.com/

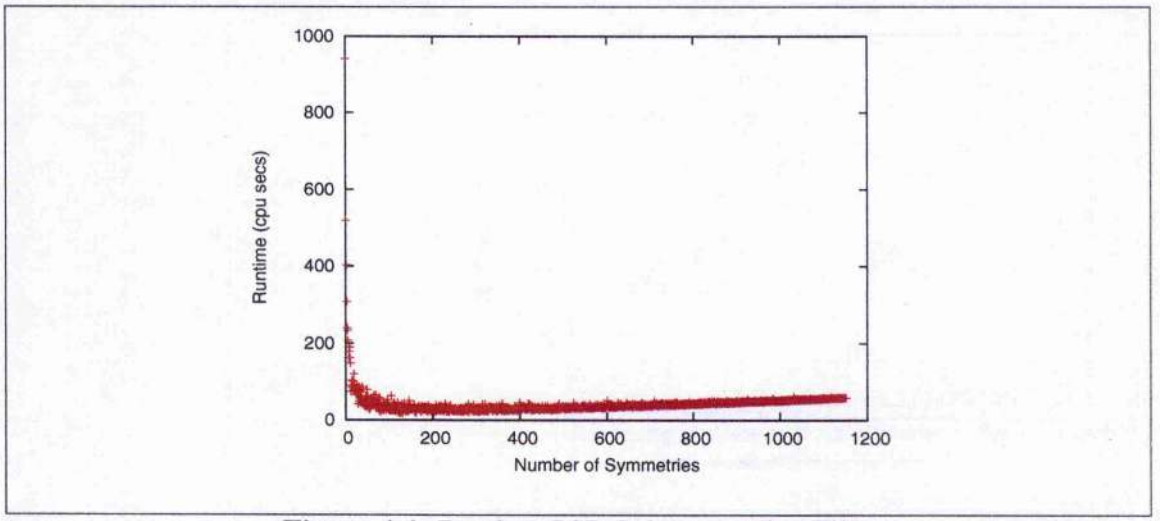


Figure 4.4: Random PSB Subsets - Alien Tiles

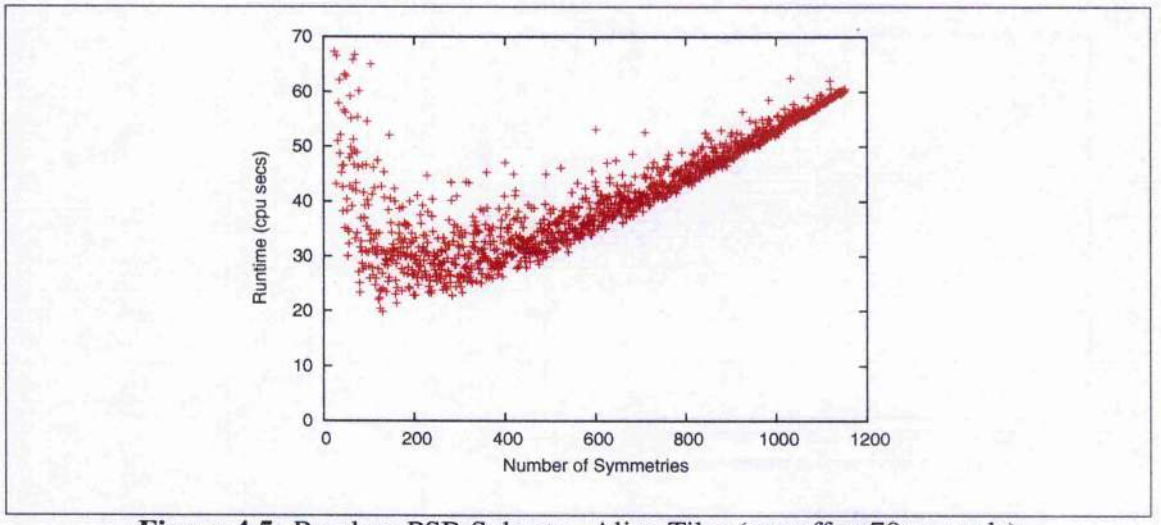


Figure 4.5: Random PSB Subsets - Alien Tiles (cut-off at 70 seconds)

problem to solve, but with $n = 4$ we can cope with all 1151 symmetry functions so we can compare PSB against breaking all symmetry. Note that the results of this research are applicable to very large groups.

Figure 4.4 shows the cpu-time to solve the alien tiles problem described above with different sized pseudo-random[4] subsets of the 1151 symmetries i.e. each point in the graph represents the runtime taken to solve the alien tiles problem with a pseudo-random subset of symmetries. Figure 4.5 shows a magnified version of a portion of the same graph, giving clearer results. By looking at the graphs we can deduce three things.

[4]In this experiment the ECL^{i}PS^e random function was used.

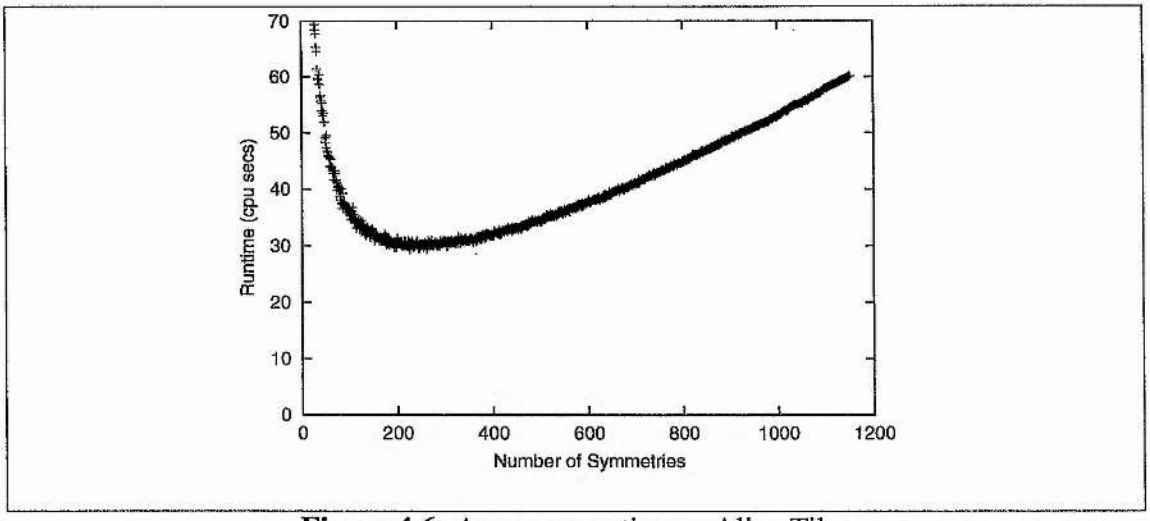


Figure 4.6: Average cpu-times - Alien Tiles

1. Most of the run-time improvement from 940.4 seconds and 18751 backtracks with no symmetry breaking to 60.5 seconds and 135 backtracks with all 1151 symmetry functions comes from adding the first 20 or so symmetries.

2. Perhaps most importantly, we can see that the shortest cpu-time comes from using a random subset of size 130. With this subset the problem was solved in 19.9 seconds and with 216 backtracks. The size of this subset is much smaller than the size of the group acting on the alien tiles CSP.

3. Different subsets of a similar size have large differences in cpu-time. This implies that the choice of symmetries we include in our subset is as important as the size of the subset. For example, another random subset of size 130 from another experiment yielded a cpu-time of 54.9 seconds (almost as much as breaking all symmetry).

The above experiment was run with 218 different random subsets[5] for each subset size to produce the less scattered curve in Figure 4.6. It is possible to gain an average factor of 2 improvement over breaking all symmetry and a factor of 32 improvement over no symmetry breaking. In the case of the subset of size 130 mentioned above, we gain a factor of 3 improvement over breaking all symmetry and a factor of 47 improvement over no symmetry breaking. The shape of the curve in Figure 4.6 is consistent with Figure 4.1 i.e.

[5]Using ECL^{i}PS^e version 5.3 on a Pentium III 1GHz processor with 512Mb of RAM

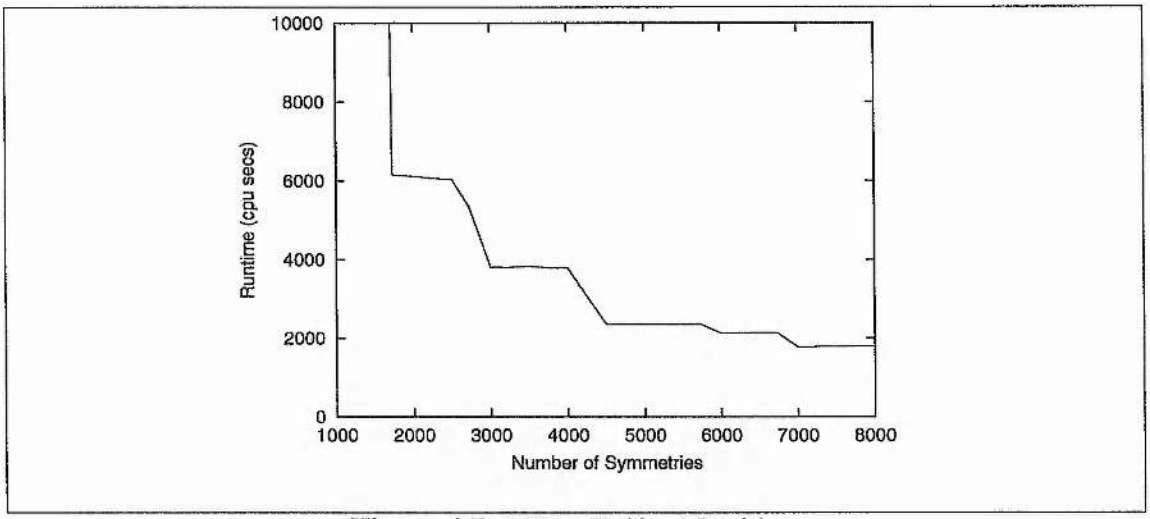


Figure 4.7: PSB - Golfers' Problem

the overhead increases approximately linearly with the number of symmetries, and there is a steep reduction in search as the first few symmetries are added. This reduction tails off as most of the redundant search is pruned, making further symmetries less effective.

4.5.3 Golfers' Problem

Here we show the existence of similar behaviour for a different problem. This uses Smith's encoding of the golfers' problem [Smi01] with $p!$ symmetries for p players. The graph shown in Figure 4.7 shows the results of finding all solutions to $golf(12, 4, 2)$[6]. Using PSB while performing a complete traversal of the search tree will yield symmetrically equivalent solutions. Smith's model has 12! or 479,001,600 symmetries. Using GAP [GAP03] it is possible to produce random elements of the group acting on this problem: S_{12}. We used GAP to output a random subset of 8000 functions representing 8000 random elements of the group. The same experiment described at the start of this section was run, with just one random subset. Due to the complexity of this problem the subsets of symmetries incremented in size in steps of 250. It was not possible to solve the problem with 1500 symmetry breaking functions within 1000 minutes of cpu-time.

The graph in Figure 4.7 is not as clear as that seen in Figure 4.6. However, whereas the alien tiles problem needed roughly 20 symmetries to do most of the symmetry breaking, the golfers' problem needs roughly 4500. We need to consider at least 1775 symmetries to be

[6]Using Ilog Solver version 4.4 on a Pentium II 300MHz processor with 512Mb RAM

able to solve this problem in reasonable time and the more symmetries we add the smaller the improvement in cpu-time. Using SBDS we are limited by the number of functions we can compile. In this respect it is more advantageous to represent symmetries using groups so that larger subsets of symmetries can be used as discussed in Chapter 4.4.1.

4.6 Partial Symmetry Breaking with Implicit Symmetry Representation

One of the nice features of using SBDS to perform PSB experiments is that there is always an explicit list or representation of the symmetries to break. Limiting this number of symmetries in order to perform PSB simply means removing some symmetries from that list.

As the more modern symmetry breaking systems are using group theoretic techniques (or other techniques that contain implicit symmetry representations), it is harder to see how to limit the symmetries in order to perform PSB. These modern approaches use a set of generators that are at very most $log_2|G|$ in size for a group G with $|G|$ elements, and in practice are much smaller (usually 2 to 6). It would be impractical to specify a list of group elements to break due to its size as well as defeating the purpose of the implicit structure. To limit the amount of symmetry breaking done we must modify the way symmetry breaking systems work. This leads to more complicated metrics as to how much symmetry is broken since we can no longer say specifically "experiment k broke n symmetries". Modifications are necessary however, in order to perform PSB with group theoretic symmetry breaking systems and other systems with implicit symmetry representations.

There are three main ways in which we can perform PSB with an implicit symmetry representation:

1. Do not perform the symmetry breaking technique at every node.

2. Use a subgroup of all symmetries.

3. Limit the amount of computation the symmetry breaking technique performs at every node.

4.6.1 Limited use of Symmetry Breaking Technique

This is a very simple method of performing PSB. Once a certain depth in the search tree is reached, we no longer execute the symmetry breaking technique [Pug03]. It is also simple to apply the symmetry breaking technique only at certain nodes in search [FSS01].

In two ways, performing PSB using this method is a good idea. Firstly, generally speaking, there is more computation involved performing symmetry breaking further down the search tree. The symmetry breaking constraints posted by SBDS become longer and thus weaker i.e. they become a lot easier to satisfy. The dominance check in SBDD is equivalent to subgraph isomorphism, whereby we are trying to map (via some symmetry) any p items of the set Q to match the set P (where $|P| = p$ and $|Q| = q$). As $|Q|$ gets larger, the problem gets exponentially more difficult and in the case of SBDD, the size of Q is the same as the number of search decisions made.

Observe that nearer the leaf nodes, symmetries become harder to break, and that also those symmetries that are broken do not prune as much search as those symmetries broken nearer the root. It is therefore clear that limiting a symmetry breaking technique to a certain depth in search is a good way of performing PSB.

4.6.2 Using a Subgroup of Symmetries

Implicit symmetry representations store a small subset of all possible symmetries. These symmetries can be repeatedly applied to each other to recreate the entire group of symmetries. As has been mentioned before, this subset is called a *generator set.* If this set has no redundant generators, then removing a generator from the set will create a group of at least an integer factor smaller. For example, consider the symmetric group S_n which has a generator set of two elements: $(1, 2)$ and $(1, 2, ..., n)$. Though this group has $n!$ elements, by removing the first generator we create a subgroup with n elements. By removing the latter we create a group with 2 elements. In general, removing just one generator can reduce the

size of the resulting group drastically.

For solving instances of the golfers' problem in [FSS01], the dominance check for eliminating all permutations of the groups, weeks and players is computationally too expensive to use during search. Therefore a dominance check for just permutations of groups and weeks is used.

The STAB technique [Pug03] uses stabilizers (which are subgroups) of the original group of symmetries during search. For this reason, the technique does not break all symmetry. Whereas other techniques can use these modifications to perform PSB to reduce runtimes, the unmodified STAB technique already performs this type of PSB.

4.6.3 Limited computation of Symmetry Breaking Technique

Finally, the amount of computation or the number of symmetry breaking constraints can be given some upperbound. This could be done by giving a time limit to every dominance check, or by stating that a maximum of n symmetry breaking constraints may be posted after any backtrack.

The SBDD dominance check implemented by Pearson in [Pea03] performs a computationally easier check by trying to map the current state into previously failed states at the same depth. Whereas the original SBDD check was equivalent to subgraph isomorphism, by only considering states with decisions the problem is reduced to graph isomorphism. Though subgraph isomorphism is known to be NP-complete, the complexity of graph isomorphism is unknown and in practice it is quite often tractable. This modification means all previous nogoods must be stored but still breaks all symmetry.

Another symmetry breaking system with an implicit symmetry representation that uses this method of PSB is STAB. By only considering the stabilizer of decisions made, the number of symmetries to consider quickly becomes smaller. However, at the root, all symmetries must be considered. STAB overcomes this problem by selecting a polynomial subset of available symmetries i.e. limiting the computation of the symmetry breaking technique. Again, it is the unmodified version of the STAB technique that performs this type of PSB.

Notice that STAB has been used to greatest effect with all three forms of PSB used simultaneously. At this time, STAB has the most impressive runtimes for solving BIBDs with

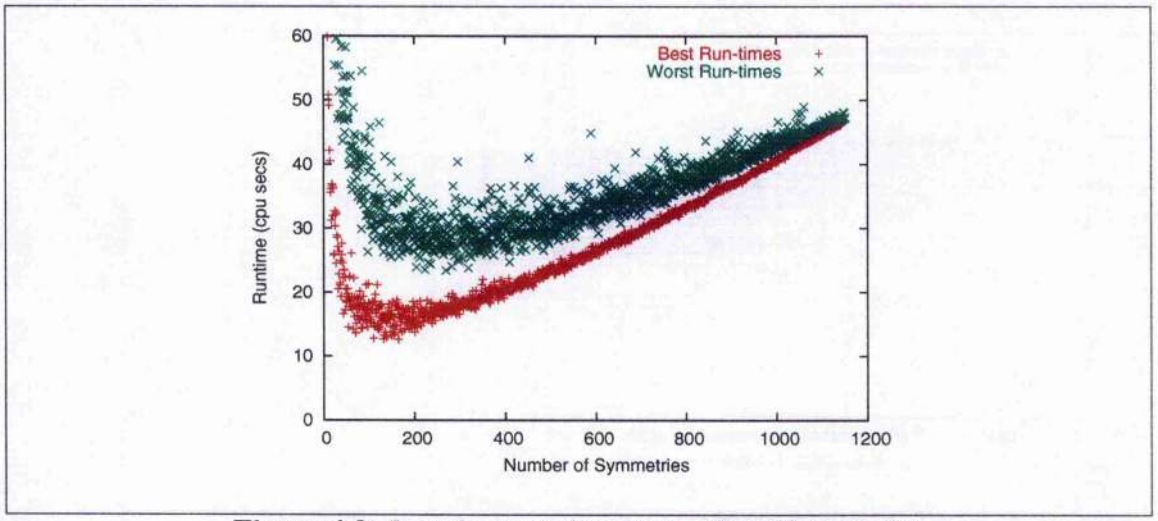


Figure 4.8: Best & worst times (cut-off at 60 seconds).

constraint programming and this helps to demonstrate the importance of PSB even with implicit symmetry representations.

4.7 Symmetry Subset Selection

In Chapter 4.5 we saw empirical evidence that using PSB can produce significant improvements. This also highlighted the importance of symmetry subset selection i.e. how we choose the subset of symmetries to break. Figure 4.8 shows the best and worst cpu-time for different sized subsets of symmetries and Figure 4.9 shows the absolute difference between them based on 15 random subsets used in the experiment[7] (described in Chapter4.5.2). The minimum cpu-time we can achieve is 12.61 seconds with a subset of 164 symmetries. However choosing a subset of this size can result in a cpu-time as large as 35.27 seconds. We now look at how the symmetry subset selection affects search and in doing so, hope to find an algorithm to select efficient symmetry subsets.

4.7.1 Looking for good subsets

In order to find an automated process for choosing sets of symmetries, we must try to look for some property that good symmetries (or sets of symmetries) have. We can then order

[7]Using ECL^{i}PS^e version 5.3 on a dual Pentium III 1GHz processor with 4Gb RAM

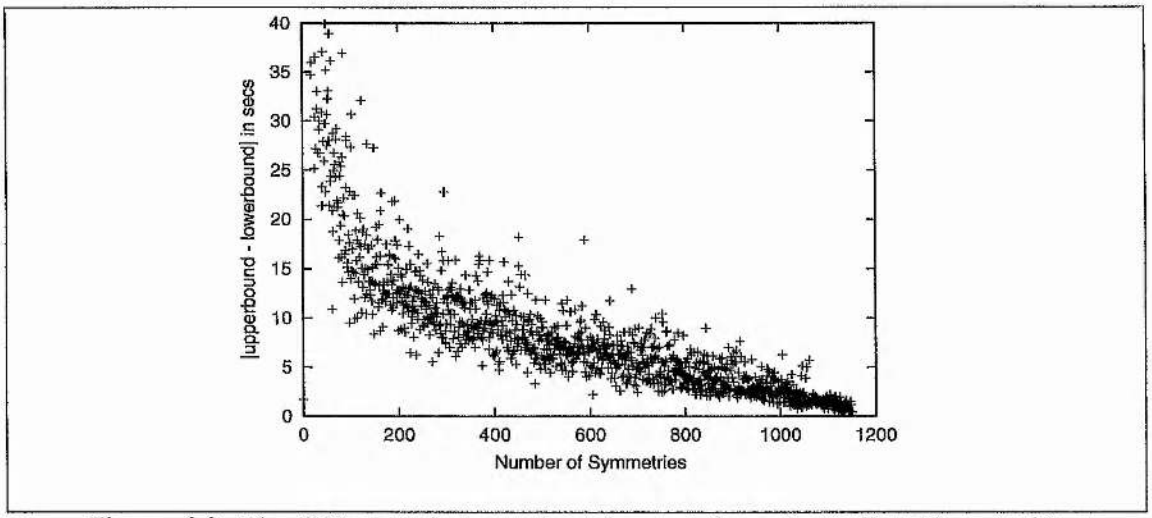


Figure 4.9: The difference between best & worst times (cut-off at 40 seconds).

the symmetries with respect to this property and choose the symmetries nearest the front of the list. The large number of experiments performed in the previous section leave a substantial amount of test data to examine in order to find such a property.

Good symmetries

The alien tiles problem was solved 251,136 times in total (218 $\times$ 1,152). The fastest 13 runs were found and the symmetries which were broken from each of these runs were recreated via the seed recorded for the pseudo-random number generator. Of the 13 subsets, the smallest was of size 97 and the largest was 230.

If there is such a thing as a good symmetry we would expect to see the same symmetries recurring in these 13 subsets. We try to find such symmetries by calculating the intersection of the subsets. However, the intersection of the first three subsets contained two symmetries and the intersection of this and the fourth subset was empty. This suggests that there are not specific symmetries that work well but rather sets of symmetries that work well together.

Patterns in symmetries

We will now try to see if there exists a pattern in good subsets of symmetries. This will be done initially by looking at small subsets (of size 8) and examining the structure of the individual symmetries. Using GAP, each symmetry is broken into individual row trans-

Col	Row	Flip
1	2	true
2	3	true
3	2	false
1	0	false
2	3	true
3	3	false
1	3	false
1	2	true

Figure 4.10: This subset took 140.05 seconds to solve the alien tiles problem.

positions, column transpositions and a possible rotation of the grid about the diagonal. Hopefully some pattern may arise in terms of how many column and row permutations a symmetry has and whether or not it contains a flip around a diagonal.

To do this, the GAP function `Factorization()` was used. This function takes two parameters, a group and an element of that group. GAP then factors the element of the group in terms of the generators. To find out how many column or row swaps etc. took place, each possible row and column swap was included as a generator as well as one which rotates the grid around the $f(x) = x$ diagonal. However, this led to unexpected behaviour from GAP. Whereas the documentation claims to return "a short word" i.e. a short list of the generators needed, the output from this did not take advantage of the redundant generators supplied and in short did not return the simplest / shortest factors. The GAP development team were contacted about this and although they say it was not a 'bug', the code would be changed (after the next release) to return the shortest factorization.

Using modified GAP code to find the shortest factorization[8], it was possible to extract the required data i.e. for any given symmetry, how many row and column transpositions is it made up of.

As can be seen by looking at Figure 4.10 and Fig 4.11, the results do not lend themselves to any immediate conclusions. This is also true of the other 13 subsets examined.

[8]Written by Steve Linton.

Col	Row	Flip
3	1	false
2	3	true
2	2	true
3	3	true
1	2	false
2	2	false
3	2	false
2	1	true

Figure 4.11: This subset took 69.05 seconds to solve the alien tiles problem.

4.7.2 The Effect of Different Heuristics

We now look to see if external factors (in this case the heuristics) are important when it comes to selecting subsets of symmetries.

When solving a symmetric CSP using symmetry breaking techniques there are two types of failure resulting in backtracking. We either fail where we discover a new unique nogood, or we fail where we find a nogood symmetrically equivalent to a previously unique nogood.

Definition 4.3 *Given a complete traversal of the search tree of a CSP L, a list of nogoods found K and a group of symmetries G (acting on L), consider a node in search k which is a nogood. If while traversing the search tree we reach node k and $\nexists g \in G$ s.t. $g(k) \in K$ then we call k a* unique nogood. *Conversely, if $\exists g \in G$ s.t. $g(k) \in K$ where $g \neq e$ (the identity element) then we call k a* symmetric nogood. *Unique nogoods result in unique fails and symmetric nogoods result in symmetric fails.*

It is straightforward to see that exponentially symmetric problems can have significantly more symmetric fails than unique fails. By performing symmetry breaking we can eliminate symmetric fails, however if we use PSB some symmetric nogoods persist. Different symmetries can be used to prune different parts of the search tree. The variable and value ordering heuristics and the propagation level dictate how the search space is traversed, therefore the symmetric nogoods pruned are dependent not only on how many symmetries we break but also on the heuristics we use.

In Figure 4.12 we present experimental evidence of this by performing the same experiment

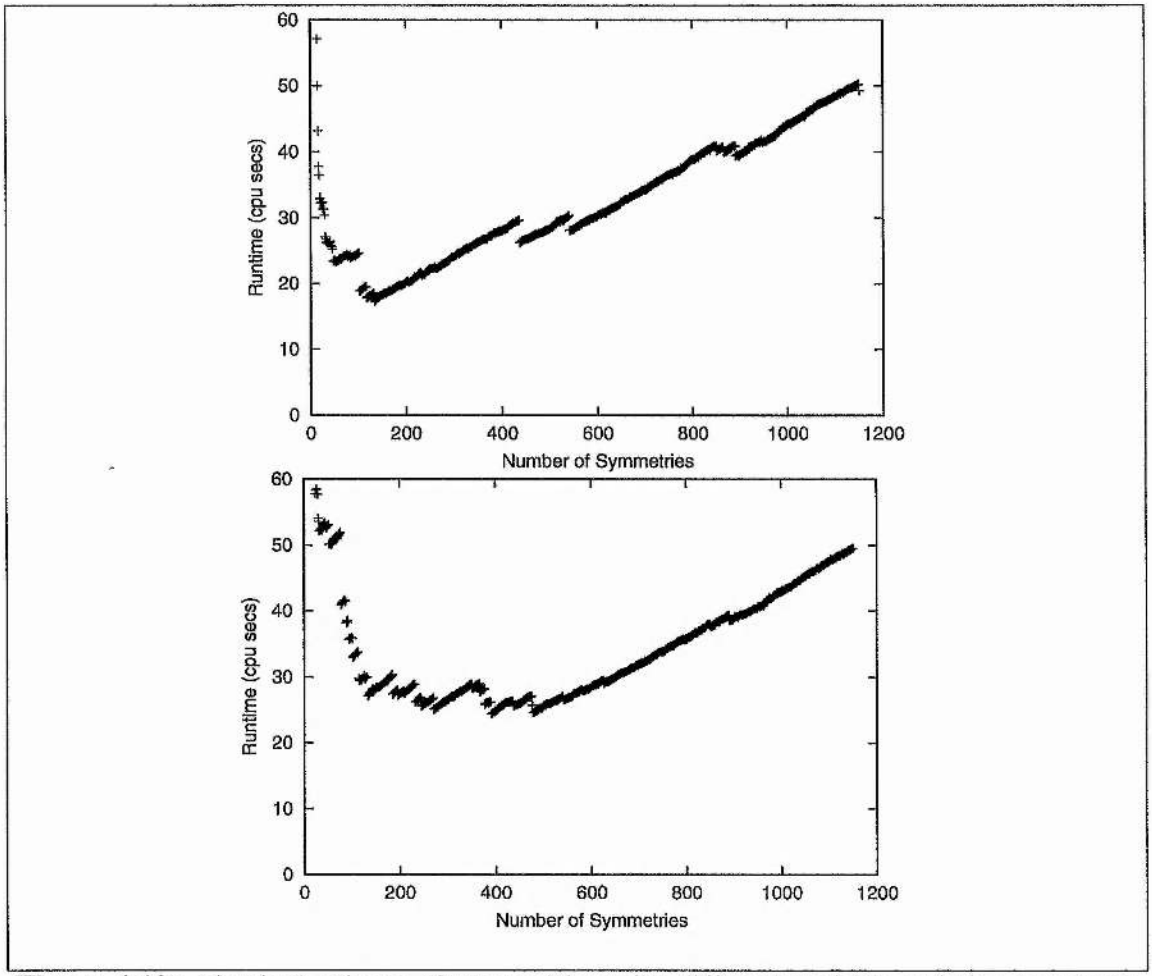


Figure 4.12: Identical subsets of symmetries with different variable ordering heuristics (cut-off 60 seconds). The first heuristic (top) is better up to 376 symmetries after which the second heuristic (bottom) takes less time

as in Chapter 4.5.2 with the same subsets of symmetries, but with different variable ordering heuristics[9]. The subsets used in Chapter 4.5.2 were randomly chosen for each different size, but in this section we have a standard subset that has a random symmetry added to it at the start of each run. The first heuristic (on the top in Figure 4.12) instantiates the alien tiles squares along the rows from top left to bottom right. The second (on the bottom) instantiates the squares along the rows from bottom right to top left. The resulting cpu-times are generally significantly different. On the other hand, changing the value ordering heuristic in solving alien tiles problems makes no difference to the number of symmetric fails we find, since the symmetries in this problem act on just the variables and not the values.

If we want to use PSB with a given symmetry breaking method we need to be aware of the variable and value ordering heuristics when we select a subset of symmetries. We can exploit this fact by choosing heuristics that work well with respect to a subset of symmetries.

4.8 Algorithm for Symmetry Subset Selection

If we wish to prune as much search as possible, it is preferable to post symmetry breaking constraints (or equivalent) that make cuts nearest the root of search. The subtree pruned by a symmetry breaking constraint is determined by 3 factors.

1. The (partial) assignment that failed

2. The symmetry being used to construct the constraint

3. The heuristics being used

Essentially, the (partial) assignment and the symmetry taken together construct the symmetry breaking constraint. The subtree this constraint forbids is determined by the heuristic. For example, consider a failed assignment $V_1 = 1$ and the symmetry g, where $g(V_1 = 1) =$

[9]Using ECL^{i}PS^e version 5.3 on a dual Pentium III processor 1GHz with 4Gb RAM

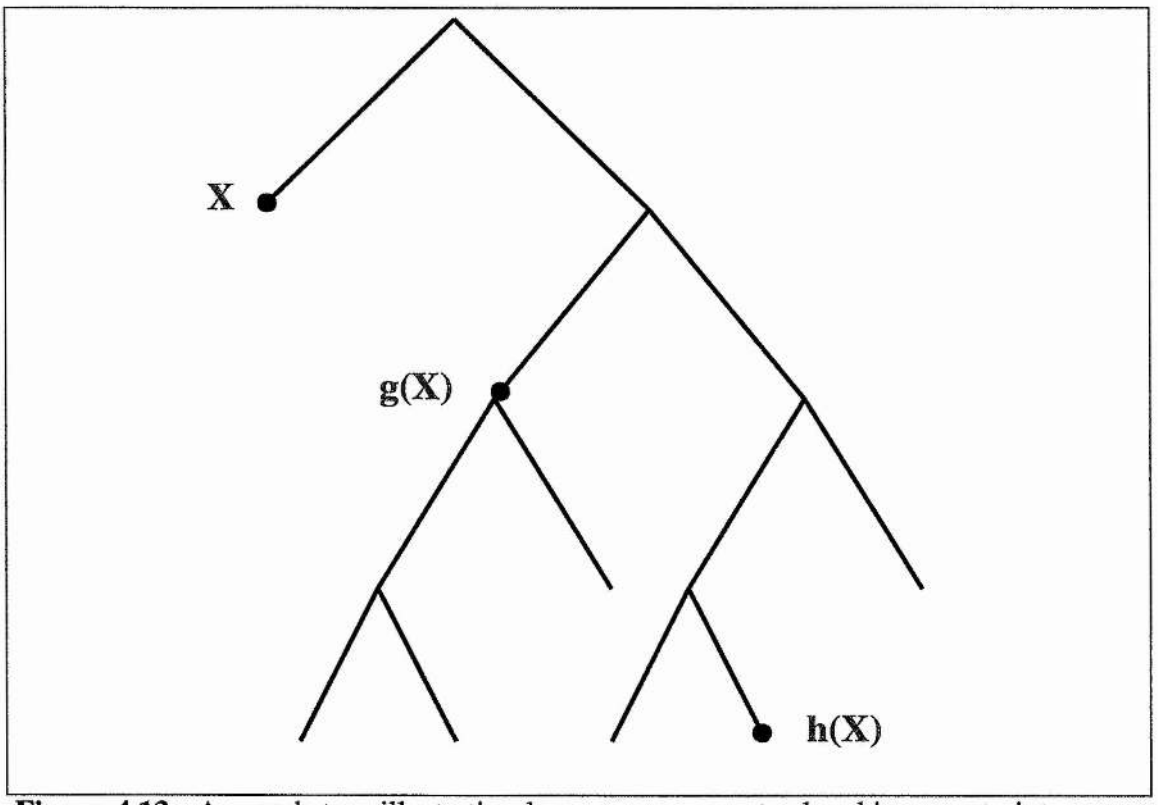


Figure 4.13: A search tree illustrating how some symmetry breaking constraints prune more search than others. Given the nogood X, g is a better symmetry to break than h.

$(V_9 = 1)$. If V_9 is the next variable to be instantiated, we can say that g, *in this case*, is a good symmetry to use. However if V_9 is one of the last variables to be instantiated, g would not be a good symmetry to use here. Another simple example of this is shown in Figure 4.13.

If we are to choose a subset of symmetries we should choose those symmetries that map the smallest (partial) assignments that will be visited by the ordering heuristics, to the root of the subtrees nearest the root according to the ordering heuristics.

To this end, we have constructed an algorithm (Algorithm 4.8.1) that takes a CSP model, a static ordering heuristic and the symmetries of the problem. This algorithm then considers every partial assignment the solver would and applies every symmetry to it, which results in a potential nogood η. We associate each η we find with the symmetry that created it and order the list of nogoods from those nearest the root (and leftmost) to those nearest the leaves (and rightmost). We then remove the duplicate symmetries from the list and we are left with the best symmetries at the front.

Algorithm 4.8.1: SYMMETRYSUBSETSELECTION($Group$, $PartialAssignments$)

for each $g \in Group$
do {
  for each $pa \in PartialAssignments$
  do {
    $element.partial\_assignment \leftarrow g(pa)$
    $element.symmetry\_used \leftarrow g$
    $element.latest \leftarrow$latest point in search in $g(pa)$[a]
    $SymmetricPartialAssignments.add(element)$
  }
}
for each $i \in SymmetricPartialAssignments$
do {
  $best \leftarrow i$
  for each $j \in SymmetricPartialAssignments$
  do {
    if $j.latest < best.latest$
    then $best \leftarrow j$
  }
  $Symmetries.add(best.symmetry\_used)$
  $SymmetricPartialAssignments.remove(best)$
}
$Symmetries.remove\_duplicate\_symmetries()$
return ($Symmetries$)

[a]Latest $var = val$ assignment to be considered by the heuristics in the partial assignment $g(pa)$.

The time complexity of Algorithm 4.8.1 is $\mathcal{O}(|G|d^n)$ for a CSP with n variables with domains of size d and a group of symmetries G acting on assignments. Thus, this algorithm is quite impractical. However, the fact that the symmetries of the alien tiles problem acts on variables means that rather than considering all $\mathcal{O}(d^n)$ partial assignments, we only need to consider the variables involved in those partial assignments. There are only n of these: $\{X_1\}$, $\{X_1, X_2\}$, ..., $\{X_1, ..., X_n\}$. The results of solving the alien tiles problem using the first k symmetries from the list of symmetries ordered using Algorithm 4.8.1, can be found in Figure 4.14.

A comparison can be found in Figure 4.15, to show that the algorithm has matched the best set of symmetries found already. The alien tiles problem can now be solved in 10.95 seconds with a subset of 92 symmetries. Algorithm 4.8.1 took only 2.11 seconds to completely order all 1,152 symmetries. The results of this experiment show that the intuition behind choosing good subsets of symmetries is correct.

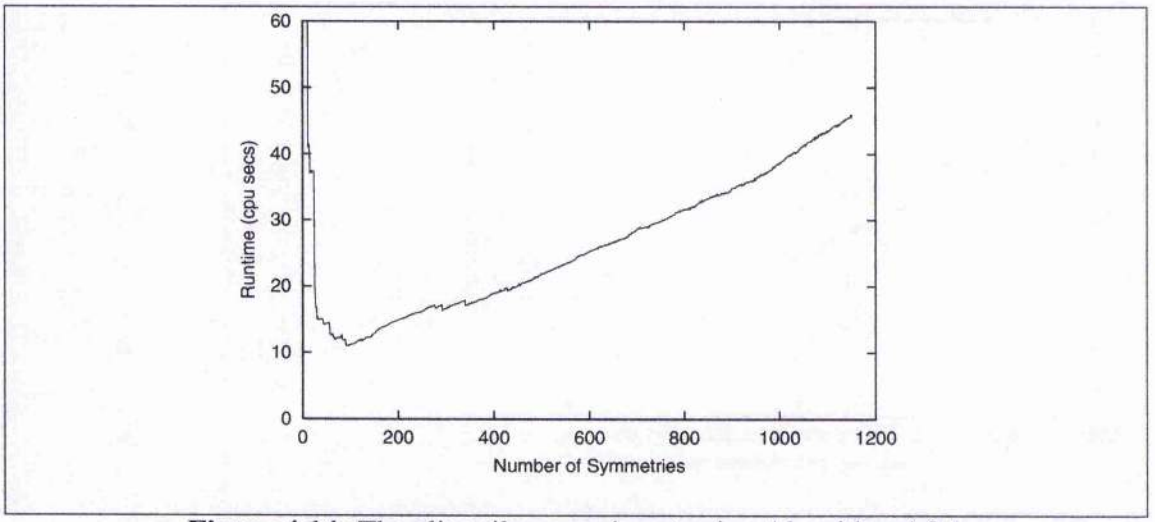


Figure 4.14: The alien tiles experiment using Algorithm 4.8.1

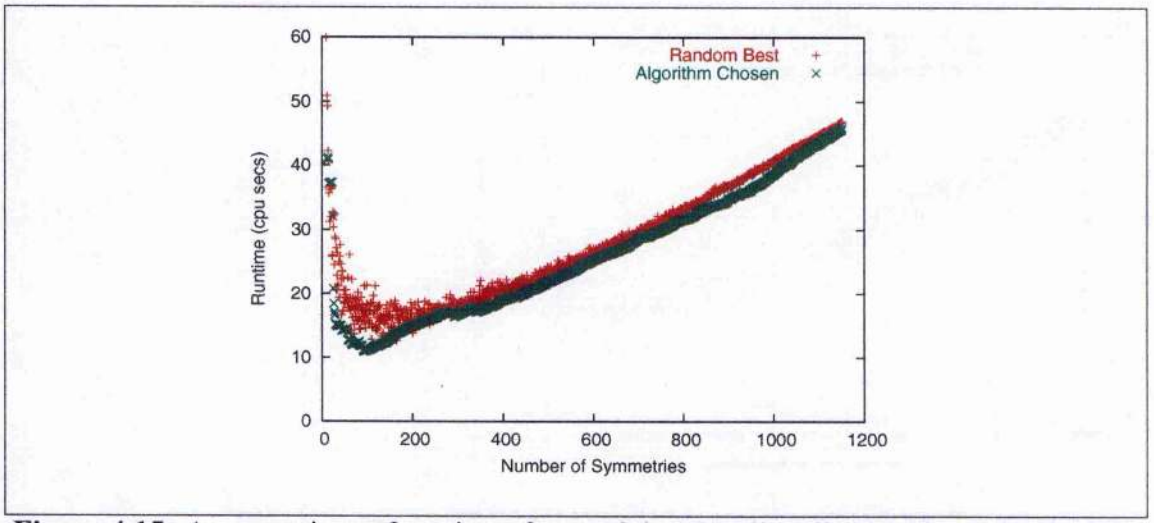


Figure 4.15: A comparison of runtimes from solving the alien tiles problem with the best random symmetries and those found using the Algorithm 4.8.1

Since a failed partial assignment with k decisions will make a symmetric nogood that will prune at depth k or greater, we can limit the algorithm to small partial assignments. By doing so we not only dramatically reduce the complexity of the algorithm but the symmetries found will still contain those that prune the most search. We also need to reduce the number of symmetries to be considered before the algorithm can be considered as practical. By selecting the setwise orbit of the partial assignment, we can drastically reduce the complexity of the algorithm further to just $\mathcal{O}(|orbit(G, A_k)|d^k)$. The value of k need only be raised until the algorithm produces a list containing as many symmetries as the constraint programmer wishes to use. For example, if $k = 5$ produces a list of $1,000$ symmetries and the constraint programmer wishes to use 900 symmetries, there is no need to rerun the algorithm with $k = 6$.

Using $k = 4$ we obtained a list of $11,880$ symmetries for the golfers' problem instance used in Chapter 4.5.3. The experiment involving the golfers' problem was repeated with the symmetries taken from the list of ordered symmetries (Figure 4.16). For a subset of $8,000$ symmetries, we were able to solve the problem in just 128.1 seconds compared to 1802.53 seconds with $8,000$ randomly chosen symmetries. The modified algorithm took only 34.8 seconds to find the best $11,880$ symmetries. The combined runtimes of the constraint solver and Algorithm 4.8.1 yield a factor of 11 improvement over the $8,000$ random symmetries chosen. For problems with larger amounts of symmetry, PSB becomes unavoidable and therefore choosing the right symmetries to break becomes more important. For larger (more symmetric) problems using the right symmetries will probably result in larger factors of improvement.

4.9 Dynamic Algorithm for Symmetry Subset Selection

Though symmetry breaking methods are mainly used to help find optimal solutions or all solutions (as we did with the alien tiles and golfers' problems above), they can still help reduce the search needed for finding the first solution. However, Algorithm 4.8.1 sorts symmetries so they are good with respect to the search tree as a whole. Therefore, the ordered symmetries are meant to be used only for finding all solutions or an optimal solution.

For example consider the search tree in Figure 4.17. The symmetries that result in the cuts

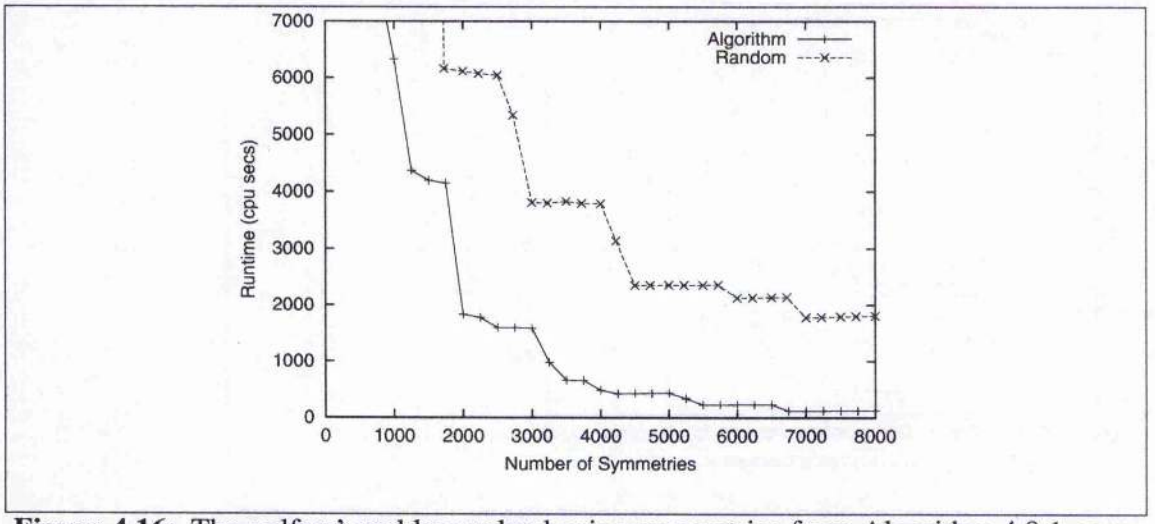


Figure 4.16: The golfers' problem solved using symmetries from Algorithm 4.9.1 compared with the original randomly chosen symmetries. We see a significant improvement in runtime even after the time for sorting symmetries is taken into account.

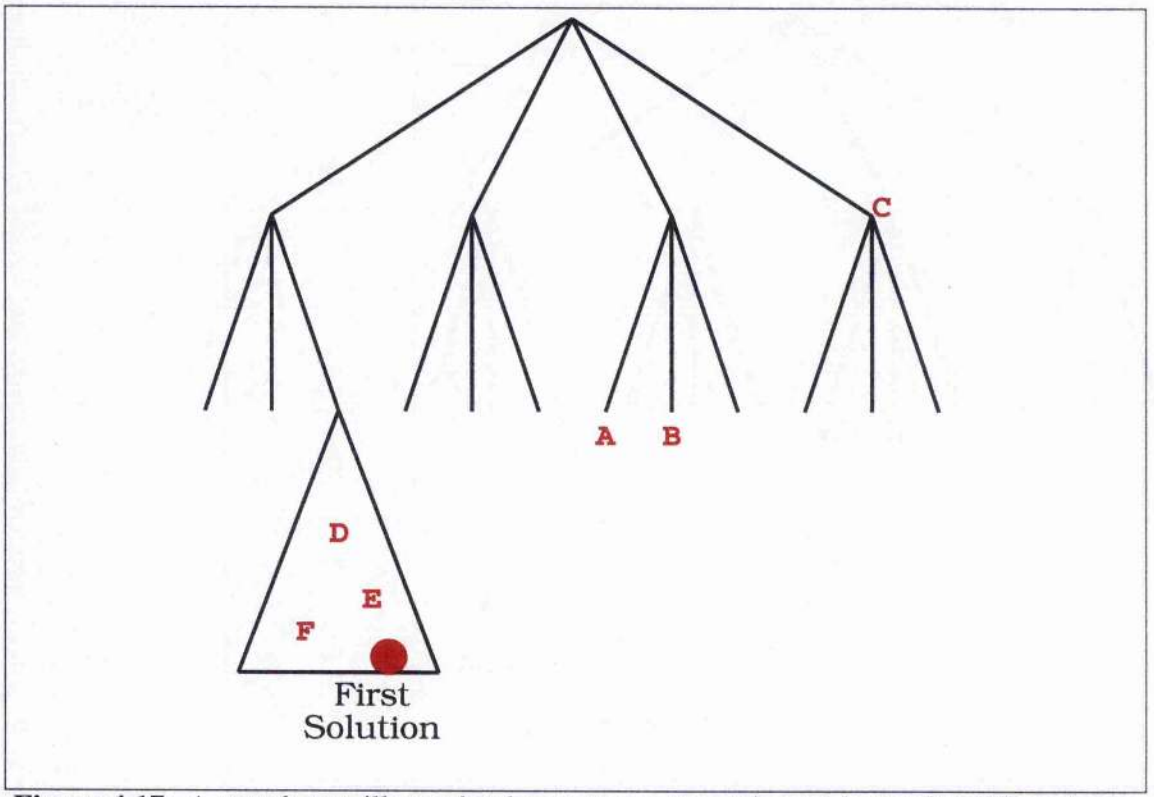


Figure 4.17: A search tree illustrating how some symmetries are better for different subtrees of search

at C, A and B would be preferred to those symmetries that cut D, E and F according to Algorithm 4.8.1. However, given the location of the first solution, we would never consider nodes C, A, or B. In order to select the best symmetries for a given subtree and thus allow effective symmetry subset selection for finding the first solution, we must sort symmetries dynamically during search.

Algorithm 4.9.1: SYMMETRYSUBSETSELECTION($Group, A$)

for each $g \in orbitRepresentatives(Group, A)$
do {
  $element.partial\_assignment \leftarrow g(A)$
  $element.symmetry\_used \leftarrow g$
  $element.latest \leftarrow$latest point in search in $g(A)$[a]
  $SymmetricPartialAssignments.add(element)$
}
for each $i \in SymmetricPartialAssignments$
do {
  $best \leftarrow i$
  for each $j \in SymmetricPartialAssignments$
  do { **if** $j.latest < best.latest$ **then** $best \leftarrow j$ }
  $Symmetries.add(best.symmetry\_used)$
  $SymmetricPartialAssignments.remove(best)$
}
$Symmetries.remove\_duplicate\_symmetries()$
return $(Symmetries)$

[a]Latest $var = val$ assignment to be considered by the heuristics in the partial assignment $g(A)$.

We now present a modified algorithm (Algorithm 4.9.1) that finds the best symmetries to consider based on just one failed partial assignment. This results in an ordering of symmetries that are optimal for the current subtree in search. As well as sorting symmetries correctly for finding the first solution, this also gives us the advantage of removing the d^n term from the complexity of Algorithm 4.8.1 to just $\mathcal{O}(|orbit(G, A)|^2)$. This complexity comes from sorting the list of symmetries after they have been found. Note that for the sake of simplicity, the algorithm encompases a bubblesort sorting algorithm. The implementation of this algorithm while semantically the same, contained a mergesort sorting algorithm. While the dynamic algorithm will still be called a potentially exponential number of times (i.e. after each backtrack), it will still be less than d^n times. The implementation and

empirical evaluation of this algorithm is left as future work.

One area the constraint programmer should note is the change in PSB metric. As mentioned previously, using a symmetry breaking system such as SBDS results in clear indication of how much symmetry was broken. It is simply the number of symmetry breaking functions used. The number of symmetry breaking constraints posted after each backtrack are in general significantly less than the number of symmetries available.

4.10 Symmetry Subset Selection for other Symmetry Breaking Systems

By looking at how good symmetries are compared to ordering heuristics and just one partial assignment, we have eliminated the exponential runtime of the symmetry subset selection algorithm. This has also given the advantage that symmetries do not need to be sorted before starting search.

Most importantly of all though this has meant that Algorithm 4.9.1 is applicable to symmetry breaking systems with implicit representations. By only sorting symmetries during search (as the symmetry breaking constraints are being posted) we can choose just the constraints that are most likely to prune the most search. Algorithm 4.9.1 is particularly relevant to both the GHK-SBDS [GHK02] and STAB [Pug03] methods and would help them solve problems with larger symmetry groups.

However, SBDD approaches are not looking for a *good* symmetry but a *specific* symmetry that satisfies the condition of dominance. In this respect, the idea behind the symmetry subset selection algorithm could possibly be used to try to find a symmetry that is more likely to find dominance. At this time though, the best methods for performing PSB with SBDD techniques are those described in Chapter 4.6

4.11 Conclusions and Future Work

In this chapter we have looked at the effect of breaking less symmetry than is possible. We defined the term partial symmetry breaking (PSB) and gave an overview of the many past

examples of PSB. We have constructed experiments that illuminate the behaviour of runtimes when solving problems with all, some or no symmetries broken. These experiments have shown that while breaking all symmetry can improve runtimes, we can solve problems even quicker by breaking a fraction of all available symmetry. The experiments also showed that different subsets of symmetries with the same size can produce very different runtimes.

These observations led us to try to explain why some symmetries work better than others when performing PSB. We looked at the possibilities of certain symmetries performing well, or patterns in successful subsets of symmetry. We then reasoned that a symmetry performs well by pruning large amounts of the search tree. The amount pruned is calculated by the symmetry, the failed nogood, and the variable and value ordering heuristics.

After realising why some symmetries are better than others, we constructed an algorithm to sort symmetries in order with respect to how well they will work with a specific (static) ordering heuristic. The effectiveness and the reasoning behind this algorithm was then shown to be correct by bettering our previous best results experimentally. The algorithm was altered to be more tractable and resulted in further reduced runtimes of constraint solving, even when combined with the runtime for sorting the symmetries.

The main contributions of this chapter therefore are threefold:

1. We have shown how most of the symmetry breaking it is possible to do can be done with a small subset of all possible symmetries

2. For constraint programmers wishing to use PSB in their experiments, we have described many different methods of performing PSB.

3. When selecting a subset of symmetries to use to perform symmetry breaking (either before search or during search) we have provided an algorithm to allow constraint programmers to find the symmetries that will prune the most search. In an experiment using 8,000 symmetry breaking functions, the algorithm chosen symmetries reduce runtime by a factor of 11 (including the runtime of the algorithm itself) compared with 8,000 randomly chosen symmetries.

Though we have a much clearer understanding of why PSB works well, we should continue to research this area as many constraint programmers do not utilise PSB to its full potential.

4.11.1 Unknown or Dynamic Heuristics

Since we have shown that breaking a set of symmetries can work well with respect to a certain heuristic, any ordering of symmetries to perform PSB using Algorithm 4.8.1 will require a static ordering heuristic that is known in advance. This is a large restriction to place on constraint solving since dynamic variable ordering heuristics can be used to dramatically reduce run-times. In this case, one should examine where the greater overhead would be i.e. using a non-dynamic heuristic or by breaking symmetries that do not prune as well as others. For exponentially symmetric problems, the difficulty in solving these problems comes from the symmetry and one could arguably say that breaking the right symmetries is more important.

However, a compromise could be achieved by combining parts of both methods. If the symmetry subset selection were to be done during search and the dynamic variable ordering heuristic is known, it is always possible to find the next variable to be instantiated. Secondly, the list of the next variables to be instantiated could be estimated based on both dynamic variable ordering heuristic and the current state in search. How this would perform in practice however is unknown.

4.11.2 Finding Subsets that break all Symmetry

By lexicographically ordering an array of variables, we can eliminate freely interchangeable variable symmetry. We can also eliminate freely interchangeable value symmetry with a polynomial subset of all symmetries [BW99]. It would be interesting if there are other interesting classes of symmetry that can be completely broken with a subset of all symmetries.

More interesting would be to learn how to apply any new findings to complex groups made from direct or wreath products of other groups. It looks most likely that to break all symmetry, we may not be able to use a small subset of symmetries. The only example of a small subset of symmetries breaking all symmetries of a CSP was found by Smith [MS02].

4.11.3 Optimal PSB point

The experiment involving the alien tiles problem (See Chapter 4.5.2) showed that there is a point where the runtime of solving a CSP is minimised. At this point, trying to break more symmetry produces little extra pruning and increases the overall runtime. Breaking fewer symmetries reduces the pruning too much and again increases the runtime.

Though it has been possible to show which symmetries it is preferable to use when performing PSB, we do not know how many symmetries we should break. The only conclusion we can state is that this number of symmetries is generally a small percentage of all available symmetries.

4.11.4 Integration of Symmetry Subset Selection into Symmetry Breaking Systems

By far the most important piece of future work that should be undertaken is the integration of the dynamic symmetry subset selection algorithm in a state of the art symmetry breaking system such as GHK-SBDS or STAB.

Much of the research into symmetry breaking in constraint programming has delivered very elegant and impressive reductions in the runtimes of CSP solving for highly symmetric problems. However, many of the advancements possible are mutually exclusive. This is not the case with PSB. Even though the performance of most modern symmetry breaking systems relies heavily on PSB, by breaking the right symmetries, the performance can be improved further.

Only by considering all aspects of the symmetry breaking system, can we produce constraint solvers that can solve problems with a super exponentially large number of symmetries.

Chapter 5

Symmetry and Propagation

By using knowledge of the symmetry that exists in certain problems, we can find solutions with less computation. There have been many different methods of dealing with symmetry, that have been used with great success. Though much research has looked at reducing redundant *search*, there has been no work into redundant *propagation*. In this chapter, we argue that we shouldn't just use symmetries to avoid redundant search but to also avoid redundant computation. We use this ethos of re-using learned information to modify a popular arc consistency algorithm and significantly reduce the amount of redundant computation.

5.1 Introduction

Previous methods of utilising symmetry in CSPs have always involved trying to break it. This can be done by affecting the search routine directly e.g. SBDD informs the constraint solver when to backtrack [FSS01]. We can also effect the solver indirectly by adding additional constraints [FFH+02] [GS00] or by using heuristics that try to avoid subtrees where symmetry is prevalent [MT01].

Constraint solving however is a balance between search and inference. There are various levels of consistency that can be maintained while searching for a solution e.g. bounds consistency, forward checking, arc consistency, path consistency. There are many algorithms for enforcing these levels of consistency, most notable are the arc consistency (AC) algo-

rithms (e.g. AC-3 [Mac77], AC-6 [BC93], AC-7 [BFR95], AC-2000, AC-2001 [BR01]) that enforce AC on binary CSPs. Though AC-6++ [BR95] makes use of the fact that "constraints are symmetric", this is a specialised case for AC-6 where, A provides support for B iff B provides support for A. This is different from the type of symmetry discussed in this thesis.

There are also algorithms for generalised arc consistency (GAC) for non binary constraints e.g. GAC-Schema [BR97]. Popular global constraints (such as *all different* [Reg94], or *lex* constraints [FHK+02]) have specialised algorithms to enforce GAC efficiently.

In this chapter, we show that symmetry in CSPs has a wider effect than just on search. We show that as well as developing systems which re-use nogoods to avoid symmetric search, we can re-use *any* information gathered about a CSP. By doing so, it is hoped that constraint programmers will look at automating methods of generating and using symmetric variants of information gathered while solving CSPs.

We begin this work by looking at *propagation algorithms*, and try to use our knowledge of symmetry to reduce the runtime and/or computation of such algorithms when dealing with symmetric CSPs.

5.1.1 Definition of Symmetry

Before continuing further, we shall refine the definition of symmetry for illustrative purposes.

Definition 5.1 *Given a CSP L, a symmetry of L is a bijective function $f : A \rightarrow A$ where A is some representation of a state in search e.g. a list of assigned variables, a set of current domains etc., such that the following holds:*

1. *Given A, a partial or full assignment of L, $(A \models \tau) \Rightarrow (f(A) \models \tau)$.*
2. *Similarly, $(A \not\models \tau) \Rightarrow (f(A) \not\models \tau)$.*

where τ can be a specific condition that a state of a CSP can satisfy e.g. arc consistent, satisfiable, etc.

The central ethos behind this new definition is that we no longer think about symmetric nogoods but rather symmetric work. Anytime any reasoning or computation is done in a symmetric CSP, we should look at how we can take advantage of it. The significance of this is that before, where we considered symmetries behaving identically in terms of nogoods, we can consider them identical in terms of anything else we can model. Previous research was concerned with reusing information on nogoods to affect search.

As a preliminary step of extending our new definition of symmetry in CSPs, we now look at consistency, more specifically arc consistency. Recall from Definition 2.2 that a binary CSP L is arc consistent if all the constraints in L are arc consistent. A binary constraint C_{ij} is arc consistent iff:

1. $\forall a \in D(X_i)\ \exists b \in D(X_j)\ s.t.\ (a,b) \in \mathcal{C}_{ij}$

2. $\forall b \in D(X_j)\ \exists a \in D(X_i)\ s.t.\ (a,b) \in \mathcal{C}_{ij}$

By re-using our knowledge of symmetry in CSPs we can either hope to enforce arc consistency with less computation or perhaps enforce a higher level of consistency.

5.2 Levels of Consistency

By using symmetry to avoid symmetric nogoods, we generally add some constraint to the problem that causes the constraint solver to backtrack earlier than it would by dealing with the constraints of the original CSP. Thus, less search is performed. If we are to look at using symmetry to avoid redundant work in consistency algorithms, we need to consider whether or not we take less time computing a consistent state or we create new constraints that enforce a higher level of consistency than is possible with the original constraints of the CSP.

5.2.1 An example modification

Algorithm 5.2.1: REVISE(V_i, V_j)

$delete \leftarrow false$
for each $x \in D(V_i)$
do { **if** $\{ \nexists y \in D(V_j) \mid (V_i = x \wedge V_j = y)$ is consistent $\}$
then { $V_i \neq x$
$delete \leftarrow true$
return $(delete)$

Algorithm 5.2.2: AC-1( )

$Q \leftarrow \{(V_i, V_j) \mid \forall (i \neq j)\}$
repeat
$change \leftarrow false$
for each $(V_i, V_j) \in Q$
do $change \leftarrow$ REVISE$(V_i, V_j) \ \vee change$
until $\neg change$

Algorithm 5.2.2 contains the pseudocode for the AC-1 algorithm where V_i is the i^{th} variable of the CSP and $D(V_i)$ is its domain. The idea behind the AC-1 algorithm is that for every pair of variables V_i and V_j, we use the *revise* function (Algorithm 5.2.1) to remove values from the domain of V_i that have no support. In doing so, we discover assignments that cannot participate in a solution based on the current domains of variables. It should also be possible to rule out the symmetrically equivalent assignments that are nogood. Based on this observation, we propose a modification to the *revise* function as is shown in Algorithm 5.2.3.

Algorithm 5.2.3: SYMMETRIC_REVISE(V_i, V_j)

$delete \leftarrow false$
for each $x \in D(V_i)$
do **if** $\{ \nexists y \in D(V_j) \mid (V_i = x \wedge V_j = y)$ is consistent $\}$
then
comment: G is the group of symmetries of the CSP
comment: A is the current set of assignments made
for each $g \in$ ORBIT$(G, A \wedge V_i = x)$
do $g(A) \Rightarrow g(V_i \neq x)$
$delete \leftarrow true$
return $(delete)$

The level of consistency enforced by this algorithm is dependent on the group we choose to use. If the group we use in Algorithm 5.2.2 is the group that stabilizes the current set of assignments i.e. $\forall g \in G, g(A) = A$ and therefore $g(A)$ is $true$, then we will enforce arc consistency.

If the group we use in Algorithm 5.2.2 is the group that represents the symmetries of the problem i.e. $g(A)$ may be satisfied, false or unknown, then we will enforce arc consistency and add extra constraints to the solver. If $g(A)$ is false, the generated constraint can be discarded. If $g(A)$ is true, the resulting unary constraint would have been generated by the AC algorithm and added at a later stage of computation. If $g(A)$ is unknown i.e. it could be satisfied later, then the resulting constraint can be added to the solver. These constraints may help the solver to backtrack earlier or prune more branches of search. In this case we can say that we have enforced a higher level of consistency.

5.3 Modifying an existing AC algorithm

Though the above algorithms are correct if the symmetries used satisfy Definition 5.1, we will initially look at modifying AC-2001, the simplest encoding of an AC algorithm with the optimal lower bound of $\mathcal{O}(ed^2)$ time and $\mathcal{O}(ed)$ space where e is the number of constraints and d is the size of the largest domain. At this stage, we will try to modify the algorithm so that it still enforces AC and not AC as well as constructing symmetry breaking

constraints.

As detailed in Chapter 5.1.1, the main ethos of the new way of looking at symmetry in CSPs is to identify the information gathered and re-use it. There are two types of information gathered during the computation of AC-2001.

1. Discovering inconsistent assignments.

2. Finding support for a particular assignment.

Finding and eliminating inconsistent assignments is the main task of AC algorithms. Consider an assignment $V_a = b$. This is can be shown[1] to be (in)consistent by first assuming that $V_a = b$ is true. Then every variable V_x, that is constrained with V_a is examined to see if there exists a domain element $y \in V_x$ such that the current set of assignments combined with $V_a = b$ and $V_x = y$ does not violate the constraint C_{ax}. If no such domain element exists, $V_a = b$ is inconsistent with the current state in search and therefore b can be removed from the domain of V_a.

AC algorithms need to consider assignments such as $V_a = b$ more than once. Rather than examine the domain of variable that is constrained with V_a, we can record assignments $V_x = y$ that do not violate any constraints when considered with the current state in search and $V_a = b$. The assignment $V_x = y$ is said to act as *support* for the assignment $V_a = b$. Recording support assignments (such as $V_x = y$) is useful since all we need to do when proving $V_a = b$ is consistent w.r.t. C_{ax} is check to see that y is still in the domain of V_x. If y has been removed from the domain of V_x then we must search for the next support for $V_a = b$. Using support assignments when enforcing AC greatly reduces the number of constraint checks needed by the AC algorithm.

Thus we will re-use these gathered data like so:

1. Forbid the orbit of inconsistent assignments under the stabilizing subgroup of the current state. If we wished to introduce symmetry breaking constraints, we could

[1]This method of achieving arc consistency is not how AC algorithms work in practice. It is merely presented as an example.

construct constraints (as SBDS does) using symmetries where the symmetric equivalent of the current state in search is not guaranteed to be true.

2. When we find support, we actually find two assignments that support each other. By taking the orbit on tuples of these two assignments we find other support assignments.

Once we find an inconsistent domain value, we can remove it since it is guaranteed to violate some constraint if instantiated. The same is true of its symmetric equivalents, therefore we can remove all the elements (assignments) of the *orbit* of this domain value. Note that the symmetric variants of the original inconsistent assignment would be discovered by the AC algorithm before it terminates. Here we remove it with the hope of reducing the number of constraint checks needed.

Though the information we re-use in step two appears to be straightforward, there is an extra condition that support assignments must meet for AC-2001. When we look for support, AC-2001 stipulates that we must use the lexicographically smallest[2] domain element as the support assignment. This condition coupled with bookmarking the last known support for an assignment means that we only search through a domain at most once.

However, symmetries that permute the domain elements of variables have destructive effects on the static lexicographic domain ordering. This makes it impossible to re-use support for symmetries that effect domain elements. If however, the symmetries of the problem permute just variables then it is possible to re-use support in the following way.

Theorem 5.1 *Assume we are given assignments $X_i = x$ and $X_j = y$ where $X_i = x$ provides support for $X_j = y$ with respect to constraint C_{ij}. If $X_i = x$ is the lexicographically least support for $X_j = y$, we can say that the smallest support for $g(X_j = y)$ is $g(X_i = x)$ if g does not affect the ordering of the domain elements.*

PROOF: If we discover support $X_i = x$ for assignment $X_j = y$ with respect to constraint C_{ij}, we must infer what this means for the other domain elements of X_i.

$X_i = x$ is the smallest support for $X_j = y$ with respect to the lexicographic ordering which means that

[2]Although any other consistent static ordering could be used.

$$\nexists x' \text{ such that } x' < x \wedge (X_i = x' \wedge X_j = y) \in C_{ij}$$

Since g does not affect the ordering of the domains, we can say that the symmetric equivalent is also true.

$$\nexists x' \text{ such that } x' < x \wedge (g(X_i) = x' \wedge g(X_j) = y) \in g(C_{ij}).$$

If $g(X_j) = y$ doesn't have support with respect to C_{ij} i.e. the algorithm has not finished the first iteration of all of the domain elements, then we can say that the smallest support for $g(X_j) = y$ is $g(X_i) = x$.

If $g(X_j) = y$ already has support with respect to C_{ij}, we can reason that it is no longer a valid support since it will be lexicographically less than $g(X_i) = x$ which violates the above condition, thus we can say that support for $g(X_j) = y$ is $g(X_i) = x$. We know that the support must be lexicographically less than $g(X_j) = y$ since X_j and $g(X_j)$ are in the same orbit. If the last support for $g(X_j)$ is the same or larger than the new support for X_j, then this is a contradiction since being in the same orbit means that $X_i = x$ would already have been discarded by stating $g^{-1}(g(X_i)) = x'$, where $x' > x$, is the smallest valid support for $X_j = y$. ■

We assume that the constraint programmer produces a group representing the symmetries of the problem prior to search. This group can then be used by the modified AC algorithm. In order to maintain AC during search, we need to note that the symmetries of the problem change as assignments are made. Anytime a search decision is made e.g. variable $X_i = j$, or variable $X_y \neq z$, we must take the *stabiliser* of these decisions. In this chapter we are taking the *pointwise* stabiliser rather than the *setwise* stabiliser of decisions. This will result in the group tending toward the identity element during search sooner but allows for cheaper group theory computations.

5.3.1 Refining AC-2001

Algorithm 5.3.1, 5.3.2 and 5.3.3 contain pseudocode based on the AC-2001 [BR01] algorithm developed by Bessière and Régin. The time and space complexity for this algorithm is optimal: $\mathcal{O}(ed^2)$ and $\mathcal{O}(ed)$ respectively.

Essentially, AC-2001 enforces AC by making every constraint $C_{ij} \in L$ to be arc consistent. This is done by looking for support for every potential assignment in the CSP. The first support assignment $X_j = b$ is stored for every assignment $X_i = a$. In this case $LAST(X_i, a, X_j) = b$. Support is searched for incrementally i.e. if $X_j = b$ is no longer a valid support for $X_i = a$ then we will only look for elements in the domain of X_j that are *lexicographically later* than b. If at any point we cannot find support for any given assignment $X_i = a$, then a is removed from the domain of X_i and $\forall j\ C_{ij}$ is placed in a queue to be made arc consistent again.

Algorithm 5.3.1: MAIN($\mathcal{X}$)

$Q \leftarrow \emptyset$;
for each $X_i \in \mathcal{X}$
do {
  for each X_j *such that* $C_{ij} \in C$
  do {
    $R \leftarrow$ SYMMETRICREVISE2001(X_i, X_j)
    for each $X_k \in R$
    do {
      if $D(X_k) = \emptyset$
      then return $(false)$;
      $Q \leftarrow Q \bigcup \{X_k\}$;
    }
  }
}
return (SYMMETRICPROPAGATION2001(Q));

Algorithm 5.3.2: SYMMETRICPROPAGATION2001(Q)

while $Q \neq \emptyset$
do {
  pick X_j from Q;
  for each X_i *such that* $C_{ij} \in C$
  do {
    $R \leftarrow$ SYMMETRICREVISE2001(X_i, X_j)
    for each $X_k \in R$
    do {
      if $D(X_k) = \emptyset$
      then return $(false)$;
      $Q \leftarrow Q \bigcup \{X_k\}$;
    }
  }
}
return $(true)$;

Algorithm 5.3.3: SYMMETRICREVISE2001(X_i, X_j)

$CHANGE \leftarrow \emptyset$;
$G \leftarrow$ group acting on CSP;
for each $v_i \in D(X_i)$
do
  if LAST$(X_i, v_i, X_j) \notin D(X_j)$
  then
    if $\exists v_j \in D(X_j) / v_j >_d$ LAST$(X_i, v_i, X_j) \bigwedge C_{ij}(v_i, v_j)$
    then
      LAST$(X_i, v_i, X_j) \leftarrow v_j$;
      if G acts on **variables**
      then
        $\mathcal{S} \leftarrow$ ORBIT$(G, [(X_i, v_i), (X_j, v_j)])$;
        for each $[(X_y, v_y), (X_z, v_z)] \in \mathcal{S}$
        then
          do LAST$(X_y, v_y, X_z) \leftarrow v_z$;
    else
      $\mathcal{O} \leftarrow$ ORBIT$(G, (X_i, v_i))$
      for each $(X_k, v_k) \in \mathcal{O}$
      do
        remove v_k from $D(X_k)$;
        $CHANGE \leftarrow CHANGE \bigcup \{X_k\}$;
return $(CHANGE)$;

The main changes to AC-2001 involve taking the orbit of inconsistent domain elements and support domain elements and re-using them. As a consequence, the Algorithm 5.3.3 doesn't return a boolean indicating whether or not the domain has been reduced, but rather a set of variables whose domain *have* been reduced. Notice how in Algorithm 5.3.3, if we find support then that support is re-used. If we cannot find support and we delete a domain element then that information is re-used to make more deletions.

5.4 Experimental Results

For the experiments, a simple backtracking binary constraint solver was implemented[3] in Java. This solver takes instances from the model B [GMP+01], random binary CSP generator [FBDR96].

[3]Thanks to Christian Bessière for helping to verify its correctness.

	Original AC-2001		Java AC-2001		
	#ccks	time	#ccks	AC del.	time
<150, 50, 500, 1250>	100,010	0.05	99,968	0	1.38
<150, 50, 500, 2350>	487,029	0.16	478,062	3,224	7.78
<150, 50, 500, 2296>	688,606	0.34	677,886	3,038	11.32
<50, 50, 1225, 2188>	1,147,084	0.61	1,114,781	1,255	18.05

Table 5.1: Results of comparing the original implementation by Bessière and Régin with the new Java implementation.

To give an idea of how it measures against the original implementation of the AC-2001 algorithm, the experiments from [BR01] were re-created (see Table 5.1, which records the number of constraint checks taken, the runtime and the number of deletions by the AC algorithm). As in [BR01], 50 instances were generated and the mean values were calculated. The runtimes of the Java implementation seem to be worse, though roughly consistent across instantiations, than the original implementation. Also, the number of constraint checks is similar. All experiments were run on an Athlon XP 2200 1.8GHz processor with 512Mb of RAM.

The main problem for the applicability of using propagation in AC algorithms is that the most symmetric problems that interest the symmetry in constraint programming community contain n-ary constraints. Such constraints cannot be dealt with by a binary AC algorithm such as is presented here.

The ideal problem for these experiments is a highly symmetric problem with a direct binary CSP model i.e. not one that needs a dual or hidden variable encoding [SW99], and where the symmetries act on variables as this would allow us to re-use support. Finding *latin squares* is such a problem. A latin square is an $n \times n$ grid of numbers from 1 to n such that each number can only appear once in each row and column. In this problem, we can freely permute the rows and columns as well as inverting around a diagonal thus giving a total of $2n!^2$ symmetries.

The results for enforcing AC on an uninstantiated instance of a latin square problem are presented in Table 5.2. Since the problem is uninstantiated (unlike when we are searching for a solution), it is trivial to calculate the size of the orbit before computing the orbit itself. This allows us to implement the orbit finding algorithm very efficiently. The latin

	AC-2001		Modified AC		
n	#con. checks	runtime	size of group	#con. checks	runtime
15	100,800	0.29	3.4×10^{24}	16	0.33
16	130,560	0.35	8.8×10^{26}	17	0.45
17	166,464	0.44	2.5×10^{29}	18	0.56
18	209,304	0.64	8.2×10^{31}	19	0.66
19	259,920	0.79	2.6×10^{34}	20	0.83
20	319,200	0.94	1.2×10^{37}	21	1.02
21	388,080	1.13	5.2×10^{39}	22	1.27
22	467,544	1.59	2.5×10^{42}	23	1.52
23	558,624	1.87	1.3×10^{45}	24	1.89
24	662,400	2.10	7.7×10^{47}	25	2.29
25	780,000	2.78	4.8×10^{50}	26	2.82
26	912,600	3.22	3.3×10^{53}	27	3.60

Table 5.2: AC on uninstantiated latin squares. The predicted number of constraint checks is produced experimentally.

	AC-2001				Modified AC			
n	fails	#ccks	AC del.	time	fails	#ccks	AC del.	time
3	0	623	5	0.02	0	324	5	0.07
4	0	3,371	9	0.06	0	1,550	9	0.13
5	4	13,432	24	0.07	5	5,743	23	0.17
6	8	41,003	40	0.13	8	14,696	40	0.29
7	55	140,454	110	0.38	55	54,141	110	0.86
8	0	198,073	63	0.62	0	54,128	63	1.28
9	95	601,669	309	2.75	101	203,451	303	4.65
10	408	2,097,243	734	12.11	409	720,123	733	16.93
11	1,277	6,785,424	3,602	48.45	1,290	2,723,297	3,607	64.49
12	5,208	49,502,231	8,654	255.21	5,208	10,412,996	8,654	348.03
13	38,209	416,371,008	72,967	2465.15	38,232	100,507,570	72,942	3315.85

Table 5.3: Maintaining AC while searching for the first solution.

squares problem is underconstrained and as such no domain removals are made. Ensuring the problem is arc consistent means just finding support for each assignment. For each variable (of which there are n^2), there are $2(n-1)$ arcs i.e. variables they are constrained with. For each arc, $n+1$ checks are required to find support for all domain elements. Thus you can see how for an $n \times n$ latin square, enforcing arc consistency takes $2n^2(n^2-1)$ constraint checks. However, for the modified algorithm, once it has been shown that one arc is arc consistent, we can infer via the symmetry of the problem that *all* arcs are consistent. So for an $n \times n$ latin square, enforcing arc consistency takes $n+1$ constraint checks.

The results for maintaining arc consistency while searching for a solution (MAC) are shown in Table 5.3. In this experiment we recorded how long it took to find the first solution to the latin squares problem, as well as the total number of constraint checks and domain deletions. At every node in search we used the AC algorithm to prune inconsistent domain elements. No data is shared between nodes so support is found from scratch with every invocation of the AC algorithm.

Disappointingly, the runtimes have not improved by re-using information. This is because the runtime of the algorithm for finding the pointwise orbit of two points outweighs the benefit of a reduced number of constraint checks. The complexity of an efficient orbit finding algorithm is $\mathcal{O}(|orbit| \times g)$ where g is the number of generators of the group. In retrospect, it is hard to improve an algorithm that has a low quadratic complexity.

Though the runtimes are not promising we manage to reduce the number of constraint checks by over a factor of 4 for MAC and by a factor of over 10,000 for AC. A more detailed look at the complexity of this algorithm would be interesting to show whether or not it could be worth using in other cases. It is hoped that more constrained problems or problems with more expensive constraint checks would be improved with inference algorithms that take symmetry into account.

5.5 Conclusions and Future Work

In this chapter we proposed ways in which symmetries in CSPs can be used to make the most of gathered information. We presented a modified version of the AC-2001 algorithm which was shown to drastically reduce the number of constraint checks needed to enforce

AC on a highly symmetric problem. Most importantly though, we have shown that the effect of symmetries in combinatorial search problems stretches further than just search and we can use this fact to avoid redundant work wherever it occurs.

This is the first step into a research area with huge potential. Though the runtimes were disappointing, the large reduction in the number of constraint checks demands further research (especially into symmetric problems with expensive constraint checks). There are many paths this research could take from here.

5.5.1 Improvements to GAC algorithms

The time complexity for GAC-Schema [BR97] is $\mathcal{O}(ed^k)$ which makes the algorithm impractical for large k. Many global constraints act on all the variables in the CSP which can make enforcing GAC more computationally expensive than actually solving the CSP.

If we consider that it takes polynomial effort to find support in AC algorithms, and polynomial effort to find symmetric variants of support assignments, we can see how using symmetry might not reduce run-times. However, for non-binary constraints, finding support takes exponential effort with respect to the arity of the constraints. Finding symmetric variants of a set of k support assignments, is not exponential. Thus, re-using support and domain removals in GAC-Schema may result in a practical algorithm for general non-binary constraints in highly symmetric CSPs.

5.5.2 Support for value symmetry

For true generality we need to be able to re-use support with symmetries that act on values as well as variables. This could be overcome by using a different AC algorithm other than AC-2001 that does not need lexicographically ordered domains. A possible solution to this is to record the symmetry used when re-using support to show what the re-ordered domain looks like.

5.5.3 Use of larger subgroups to enforce higher levels of consistency

The work carried out in this thesis contained very strict restrictions when creating subgroups of symmetries. By ensuring the group we dealt with stabilized both positive and negative decisions, we enforced the same amount of consistency as a non-symmetry breaking AC algorithm.

By stabilizing just the positive decisions we conjecture that the modified AC algorithm would post the unary constraints that SBDS would post. This would achieve a higher level of consistency. It may also be possible to construct constraints like those posted by SBDS that would forbid future decisions that are inconsistent as suggested by Algorithm 5.2.3. This future work would require a theoretical evaluation to ensure that it is a valid method of breaking symmetry and does not delete consistent assignments.

5.5.4 Concise representations of constraints using group theory

Rather than describing symmetric constraints explicitly, it should be possible to introduce a new data type that consists of a constraint and a group. To use the latin squares problem as an example, the constraint would be *all different* on any one row or column. The group would be the one described in Chapter 5.4.

As well as being able to describe CSPs more easily, we could employ the ideas in this chapter to the k-consistency algorithm [Coo89]. Not only would we be able to reduce the run-time of this algorithm by using knowledge of symmetry, we would be able to reduce the space complexity by concise representations of the exponential number of constraints produced by the k-consistency algorithm.

This would involve much research into providing mechanisms for propagating implicit constraints effectively. The potential benefits however are very important in our goal of significantly reducing the time needed to solve highly symmetric problems.

Chapter 6

Conclusions and Future Work

There are many difficult problems that need practical methods of solving them. One such method for solving combinatorial search problems is constraint programming. This method of problem solving focuses on developing superior constraint solvers. Constraint satisfaction problems (CSPs) are described in general ways which means one constraint solver can be used to solve many different and complex problems. We arrive at a state where problems are easy to express, and efficient to solve.

Symmetries are commonplace in the real world. The structure we place on physical objects helps us to reason with them more easily. Symbolical artificial intelligence requires objects to be labelled explicitly, and thus they are naïve to the inherent symmetries that exist in the problems they are solving. This lack of awareness of symmetry causes redundancy in solving constraint satisfaction problems.

In some areas of computer science, some redundancy may be acceptable. The exponential complexity of solving NP-complete problems however, means that the methods employed must be as efficient as possible. Thus if we are to use constraint programming to solve problems with symmetry, we must develop efficient methods of utilising the symmetry. Indeed, without considering the symmetry of certain problems, they become impossible to solve in a reasonable time.

There are many subtle aspects of how symmetry affects solving CSPs. Similarly there exist many ways of breaking symmetry in CSPs. In this thesis we investigated various ways in which breaking symmetry while using constraint programming could be improved. This

involved looking at many of the diverse problems that symmetrical CSPs introduce.

6.1 Contributions

We now recount the achievements of this thesis. To each problem encountered, we describe how we bettered the current research.

6.1.1 Implementation of Symmetry Breaking Systems

The way we describe problems using constraint programming is separate to way the computer solves them. That is, a constraint programming language is made up of an interface for describing models and constraints, and also a hidden set of algorithms that perform backtrack search and maintain consistency etc. In a similar way, the way we describe the symmetries of CSPs should be kept seperate to the way in which they are broken. By doing so we hope to create symmetry breaking systems that easy to use. Currently, there is only one implementation of a symmetry breaking system included with a constraint solver [WNS97]. As the research into symmetry breaking in constraint programming matures, we will see more constraint solvers including symmetry breaking systems.

In this thesis (Chapter 3.1) we critiqued the advantages and disadvantages of previous symmetry breaking methods from the point of view of the constraint programmer. We then listed the ideal requirements for future implementations of these systems. This is an extremely important step it we wish to see symmetry breaking techniques being used by the wider constraint community.

6.1.2 Concise Representation of Symmetries

Prior to the research in this thesis (Chapter 3.2), all generic symmetry breaking techniques for constraint programming required an explicit representation of symmetries. We introduced the symmetry breaking system U-SBDS which uses group theory to represent symmetries concisely by only listing a generator set of symmetries (Chapter 3.2). Apart from Brown, Finkelstein and Purdom's work [BFP96] we are not aware of any generic symmetry

breaking method for backtrack search before this thesis.

Using the concise representation of symmetries that group theory gives makes the difference between being able to describe thousands of symmetries and being able to describe millions of orders of magnitude more. Without this step forward, we would not be able to deal with the amount of symmetry it is possible to today.

6.1.3 Analysis of Number of Symmetry Breaking Constraints

The symmetry breaking method SBDS [BW99] [GS00] presented a general method of adding constraints to the local subtree that would break a specific symmetry. The optimisation in [GS00] showed that symmetries that were guaranteed to be broken at a node in search could be discarded from that subtree. We saw how using the orbit finding algorithm to calculate the symmetries to break, drastically reduced the number of symmetries to consider near the root of search. A result of this is a reduced number of symmetry breaking constraints needed to break all symmetry. This was also shown in [McD01]. The research in [GHK02] combined the first optimisation and a limited version of the second. In this thesis we showed how many symmetry breaking constraints would be needed to break all symmetry for specific groups and symmetry breaking methods (Chapter 3.2.2). We argued (as in we did in [McD01]) that a symmetry breaking system that used the intersection of unique symmetries (Definition 3.1) with broken symmetries [GS00] would be able to break more symmetries than previous methods.

6.1.4 A Method for Describing Symmetries

It was shown that symmetry breaking methods either apply to specific symmetries or general symmetries. For those systems that deal with specific symmetries, they can only be used when the CSP they are dealing with has those symmetries. The symmetry breaking systems that deal with general symmetries need a method of describing those symmetries. Previously this has meant defining some permutation for each symmetry, writing another CSP representing the symmetries of the original CSP, or writing a permutation group and a method of translating from points to assignments and vice versa. We presented a method for describing symmetries that provides three main advantages over any previous method

(Chapter 3.5):

1. The symmetry breaking method is hidden and thus the same description can be used for different symmetry breaking systems.

2. By using descriptive words rather than groups or other CSPs, symmetries can be described more easily. Thus it is more accessible to the average constraint programmer unfamiliar with symmetry breaking.

3. Providing descriptions in general terms means that the symmetries scale for different sized problems i.e. the symmetries are problem specific and not instance specific.

We implemented such a system, NuSBDS, gave example code and provided empirical evidence.

6.1.5 Using Subsets of all Symmetry

We presented the first comprehensive study of breaking just subsets of symmetries i.e. PSB. We analysed how breaking just a subset of symmetries altered the performance of problem solving. Based on these experiments we noticed that not only is the number of symmetries broken important but also which symmetries were chosen. By realising that different subsets broke more symmetry than others we then looked at quantifying what made a good subset of symmetries.

We reasoned that the good symmetries were those that produced symmetry breaking constraints that pruned the highest nodes of search. We then created an algorithm that ordered symmetries based on this criterion.

We then provided empirical evidence of using this algorithm. With subsets of symmetry generated by the symmetry sorting algorithm, we gained better results than already found from the best random subsets found so far (in the best case, a factor of 11 improvement).

Many other symmetry breaking systems have the potential to be improved further by incorporating the results of this work into partial symmetry breaking.

6.1.6 Observed Redundant Computation

Though using the symmetry in CSPs can drastically reduce the size of the search tree we traverse, no research has been undertaken to reduce other redundant computation. Since constraint programming is, among other things, a balance of search and propagation, we reasoned that symmetries could be used to avoid more than redundant search. This thesis is the first research to look at other redundant computation.

In doing so, we look at a specific algorithm that contains redundant work. We recognise the computation performed and also how it can be re-used to avoid redundant computation.

Although the run-times of the particular algorithm were not lowered by adding symmetry breaking, this is just the first step in this new area. The central theme of this research can be applied to other algorithms that perform redundant work. There is a great potential is this respect for further improvements in solving symmetric problems.

6.2 Future Work

Most of the contributions to the area of symmetry in constraint programming presented in this thesis can be extended further. By doing so we can produce more efficient and general ways of solving CSPs with large amounts of symmetry.

6.2.1 Symmetry Breaking Implementations

We need to see more symmetry breaking systems being included in constraint solvers. Thus far this has been a slow process. The main reason for is the complicated nature of recognising and describing symmetries: currently a process mastered by generally just symmetry researchers. Also, describing incorrect symmetries can lead to incorrect output. This is a real problem for beginners in the field of symmetry breaking.

Even if we were to find a guaranteed optimal method for symmetry breaking, it would be hard to include it with current constraint solvers. More research is needed to look at how difficult the average constraint programmer finds the task of recognising symmetry. We also need a standard technique for describing direct and wreath products of symmetries easily and correctly.

6.2.2 An improved version of SBDS

For an exponential number of symmetries, SBDS posts an exponential number of constraints. This overhead is the main reason for the upperbound on the number of symmetries that can be broken for an SBDS-like symmetry breaking method. SBDD-like implementations have recently been the preferred method of breaking large amounts of symmetry since they do not post any additional constraints to the solver. Even though the SBDD checks take an exponential amount of time, the lack of an exponential memory requirement means that they can break larger amount of symmetry.

However, the new symmetry breaking method STAB, posts symmetry breaking constraints during search and yields good run-times in comparison to other methods. By reducing the overhead and redundancy of SBDS further, it may be possible to create a symmetry breaking method that can break a larger number of symmetry than any other. This can be done by implementing a method of SBDS that contains the advantage of discarding broken symmetries, posting unique constraints and possibly breaking the subset of symmetries that provides the best pruning to overhead ratio.

6.2.3 Dynamic Partial Symmetry Breaking

Though an algorithm to perform symmetry sorting dynamically during search is provided in this thesis, it has not been implemented. Such an implementation would theoretically be able to handle large groups by breaking just the symmetries that would prune the most search. Though the results of using sorted subsets of symmetries in this thesis are limited to SBDS, it would be interesting and worthwhile to see if they could be employed to work with other symmetry breaking systems.

6.2.4 Avoiding more Redundant Computation

It is the area of avoiding redundant computation which yields the most unexplored avenues. In this thesis we used the symmetries of a highly symmetric problem to drastically reduce the number of constraint checks needed. However, the small computational complexity of the original algorithm meant that there were no reductions in runtime.

The future work involves looking at some of the many other algorithms in constraint programming that do not take advantage of the symmetries of CSPs. Algorithms with larger complexities will most probably result in much enhanced performance by including symmetry breaking techniques.

Symmetries provide lots of benefits to the constraint programmer. We must ensure that in future, we develop methods that allow us to take advantage of these benefits to solve our problems as efficiently as possible.

Bibliography

[Agu93] Alfonso San Miguel Aguirre. How to use symmetries in boolean constraint solving. In Frederic Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 287–306. MIT Press, 1993.

[ARMS03] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. In *IEEE Trans. on CAD, vol. 22(9)*, pages 1117–1137. 2003.

[BC93] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *AAAI-93: Eleventh National conference on Artificial Intelligence*, pages 108–113, Washington, DC, 1993.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust. The MAGMA algebra system I: the user language. In *Journal of Symbolic Computation*, volume 24, pages 235–265. Academic Press, Inc., 1997.

[BCvW02] Fahiem Bacchus, Xinguang Chen, Peter van Beek, and Toby Walsh. Binary vs. non-binary constraints. In *Artificial Intelligence*, volume 140, pages 1–37. Elsevier Science Publishers Ltd., 2002.

[Ben94] Belaid Benhamou. Study of symmetry in constraint satisfaction problems. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, Orcas Island, Seattle, USA, 1994.

[BFP88] Cynthia Brown, Larry Finkelstein, and Paul Purdom. Backtrack searching in the presence of symmetry. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 99–110. Springer, 1988.

[BFP96] Cynthia Brown, Larry Finkelstein, and Paul Purdom. Backtrack searching in the presence of symmetry. In *Nordic journal of Computing*, pages 203–219. Publishing Association Nordic Journal of Computing, 1996.

[BFR95] Christian Bessière, Eugene Freuder, and Jean-Charles Régin. Using inference to reduce arc-consistency computation. In *Fourteenth International Joint Conference of Artificial Intelligence*, pages 592–598, Montreal, Canada, 1995.

[BPN01] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers, 2001.

[BR95] Christian Bessière and Jean-Charles Régin. Using bidirectionality to speed up arc-consistency processing. In M. Meyer, editor, *Constraint Processing*, volume LNCS 923, pages 157–169. Springer, 1995.

[BR97] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: preliminary results. In *International Joint Conference of Artificial Intelligence*, pages 398–404, Nagoya, Japan, 1997.

[BR01] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *International Joint Conference of Artificial Intelligence*, pages 309–315, Seattle, WA, 2001.

[BS92] Belaid Benhamou and Lakhdar Sais. Theoretical study of symmetries in propositional calculus and applications. In D. Kapur, editor, *CADE: International Conference on Automated Deduction*, pages 281–294. Springer, 1992.

[Bur97] William Burnside. *Theory of groups of finite order*. Cambridge University Press, 1897.

[But91] Gregory Butler. *Fundamental Algorithms for Permutation Groups*. Springer-Verlag, 1991.

[BW98] Rolf Backofen and Sebastian Will. Excluding symmetries in concurrent constraint programming. In *Modeling and Computing with Concurrent Constraint Programming*, 1998.

[BW99] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming*, pages 73–87. Springer, 1999.

[Cay78] Arthur Cayley. On the theory of groups. In *London Mathematics Society*, volume 9, pages 126–133. 1878.

[CB02] Mats Carlsson and Nicolas Beldiceanu. Arc-consistency for a chain of lexicographic ordering constraints. Technical Report Research Report T2001-18, Swedish Institute of Computer Science, 2002.

[CF93] Gene Cooperman and Larry Finkelstein. Combinatorial tools for computational group theory. In *Groups and Computation*, volume 11, pages 53–86, 1993.

[CGLR96] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Knowledge Representation'96: Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, San Francisco, California, 1996.

[CHS+03] Andrew Cheadle, Warwick Harvey, Andrew Sadler, Joachim Schimpf, Kish Shen, and Mark Wallace. ECL^iPS^e: An Introduction. Technical report, IC-Parc-03-1, 2003.

[CLW99] B. M. W. Cheng, J. H. M. Lee, and J. C. K. Wu. Speeding up constraint propagation by redundant modeling. In *Constraints*, volume 4(2), pages 167–192. 1999.

[Coo89] Michael Cooper. An optimal k-consistency algorithm. In *Artificial Intelligence, Volume 41(1)*, pages 89–95. Elsevier Science Publishers, 1989.

[Cra92] James Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In *AAAI Workshop on Tractable Reasoning*, pages 17–22, 1992.

[DdVC03] Iván Dotú, Alvaro del Val, and Manuel Cebrián. Redundant Modeling for the QuasiGroup Completion Problem. In F. Rossi, editor, *Principles and Practice of Constraint Programming*, pages 288–302. Springer, 2003.

[Dec90] Rina Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. In *Artificial Intelligence*, volume 41, pages 273–312. Elsevier Science Publishers Ltd., 1990.

[DF99] Rina Dechter and Daniel Frost. Backtracking algorithms for constraint satisfaction problems; a survey. Technical report, UCI Tech Report, 1999.

[Far02] Julian Faraway. *Practical Regression and Anova in R*. 2002. Available from http://www.stat.lsa.umich.edu/~faraway/book/pra.pdf.

[FBDR96] Daniel Frost, Christian Bessière, Rina Dechter, and Jean-Charles Régin. *Random Uniform CSP Generator*, 1996. Available from http://www.lirmm.fr/~bessiere/generator.html.

[FFH+01] Pierre Flener, Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Matrix modelling. Technical Report APES-36-2001, APES Research Group, 2001. Available from http://www.dcs.st-and.ac.uk/~apes/apesreports.html.

[FFH+02] Pierre Flener, Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In P. van Hentenryck, editor, *Principles and Practice of Constraint Programming*, pages 462–476. Springer, 2002.

[FHK+02] Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Global constraints for lexicographic orderings. In P. van Hentenryck, editor, *Principles and Practice of Constraint Programming*, pages 93–108. Springer, 2002.

[FL02] Maria Fox and Derek Long. Extending the exploitation of symmetries in planning. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Artificial Intelligence Planning Systems*, pages 83–91, 2002.

[FM01] Filippo Focacci and Michaela Milano. Global cut framework for removing symmetries. In Toby Walsh, editor, *Principles and Practice of Constraint Programming*, pages 77–92. Springer, 2001.

[Fre91] Eugene Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *American Association for Artificial Intelligence*, pages 227–233, 1991.

[FSS01] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *Principles and Practice of Constraint Programming*, pages 93–107. Springer, 2001.

[GAP03] The GAP Group, Aachen, St Andrews. *GAP - Groups, Algorithms, and Programming, Version 4.3*, 2003.

[Gas79] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, Pa., 1979.

[Gen01] Ian P. Gent. A symmetry breaking constraint for indistinguishable values. In Piere Flener and Justin Pearson, editors, *SymCon'01: Symmetry in Constraints*, pages 25–32, 2001.

[GHK02] Ian P. Gent, Warwick Harvey, and Tom Kelsey. Groups and constraints: Symmetry breaking during search. In P. van Hentenryck, editor, *Principles and Practice of Constraint Programming*, pages 415–430. Springer, 2002.

[GHKL03] Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton. Generic SBDD using Computational Group Theory. In F. Rossi, editor, *Principles and Practice of Constraint Programming*, pages 333–347. Springer, 2003.

[Glo89] Fred Glover. Tabu search - part i. In *OSRA Journal of Computing*, volume 1(3), pages 190–206. 1989.

[GLS00] Ian P. Gent, Steve Linton, and Barbara M. Smith. Symmetry breaking in the alien tiles puzzle. Technical Report APES-22-2000, APES Research Group, October 2000.

[GM03] Ian P. Gent and Iain McDonald. Symmetry and Propagation: Revising an AC algorithm. In Barbara M. Smith, Ian P. Gent, and Warwick Harvey, editors, *Symmetry in Constraint Satisfaction Problems*, pages 66–74, 2003.

[GMP+96] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In Eugene C. Freuder, editor, *Principles and Practice of Constraint Programming*, pages 179–193. Springer, 1996.

[GMP+01] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. In *Constraints*, volume 6, pages 345–372. 2001.

[GMS03] Ian P. Gent, Iain McDonald, and Barbara M. Smith. Conditional symmetry in the all-interval series problem. In Barbara M. Smith, Ian P. Gent, and Warwick Harvey, editors, *Symmetry in Constraint Satisfaction Problems*, pages 55–65, 2003.

[Gol02] Eugene Goldberg. Testing satisfiability of CNF formulas by computing a stable set of points. In *Symposium on the Theory and Applications of Satisfiability Testing*, pages 54–69, 2002.

[GPS02] Ian P. Gent, Patrick Prosser, and Barbara M. Smith. A 0/1 encoding of the gaclex constraint for pairs of vectors. In *W9 Modelling and Solving Problems with Constraints*, 2002.

[GS00] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In W. Horn, editor, *Proceedings of ECAI-2000*, pages 599–603. IOS Press, 2000.

[GW99] Ian P. Gent and Toby Walsh. Csplib: a benchmark library for constraints. Technical report, APES-09-1999, 1999. Available from http://www.csplib.org/.

[Har01] Warwick Harvey. Symmetry breaking and the social golfer problem. In Piere Flener and Justin Pearson, editors, *SymCon'01: Symmetry in Constraints*, pages 9–16, 2001.

[HFPA03] Pascal Van Hentenryck, Pierre Flener, Justin Pearson, and Magnus Agren. Tractable Symmetry Breaking for CSPs with Interchangeable Values. In *International Joint Conference of Artificial Intelligence*, pages 575–580, Acapulco, Mexico, 2003.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *International Joint Conference on Artificial Intelligence*, pages 607–615. Morgan Kaufmann, 1995.

[HKP03] Warwick Harvey, Tom Kelsey, and Karen Petrie. Symmetry Group Expression for CSPs. In Barbara M. Smith, Ian P. Gent, and Warwick Harvey, editors, *Symmetry in Constraint Satisfaction Problems*, pages 86–96, 2003.

[ILO00] ILOG S.A., Gentilly, France. *ILOG Solver, Version 5.0*, 2000.

[JR97] David Joslin and Amitabha Roy. Exploiting symmetry in lifted CSPs. In *American Association for Artificial Intelligence*, pages 197–202, 1997.

[Kiz04] Zeynep Kiziltan. *Symmetry Breaking Ordering Constraints*. PhD thesis, Uppsala University, 2004.

[Kum92] Vipin Kumar. Algorithms for constraints satisfaction problems: A survey. In *The AI Magazine, by the AAAI*, volume 13, pages 32–44. 1992.

[LJP02] Chu Min Li, Bernard Jurkowiak, and Paul W. Purdom. Integrating symmetry breaking into a DLL procedure. In *Symposium on the Theory and Applications of Satisfiability Testing*, pages 149–155, 2002.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. In *Artificial Intelligence, Volume 8*, pages 99–118. 1977.

[Mar03] François Margot. Exploiting Orbits in Symmetric ILP. In *Mathematical Programming*, volume Series B 98, pages 3–21. 2003.

[McD01] Iain McDonald. Unique Symmetry breaking in CSPs using Group Theory. In Piere Flener and Justin Pearson, editors, *SymCon'01: Symmetry in Constraints*, pages 75–78, 2001.

[McD03] Iain McDonald. NuSBDS: Symmetry Breaking made Easy. In Barbara M. Smith, Ian P. Gent, and Warwick Harvey, editors, *Symmetry in Constraint Satisfaction Problems*, pages 153–160, 2003.

[McK98] Brendan D. McKay. Isomorph-free exhaustive generation. In *Journal of Algorithms*, volume 26(2), pages 306–324. 1998.

[MH86] Roger Mohr and Thomas Henderson. Arc and path consistency revisted. In *Artificial Intelligence*, volume 28, pages 225–233. 1986.

[MR90] R. Mathon and A. Rosa. Tables of parameters of bibd with r≤41 including existence, enumeration and resolvability results: an update. In *Ars Combinatoria*, volume 30. 1990.

[MS02] Iain McDonald and Barbara M. Smith. Partial symmetry breaking. In Pascal van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP2002*, pages 431–445. Springer, 2002.

[MT01] Pedro Meseguer and Carme Torras. Exploiting symmetries within constraint satisfaction search. In *Artificial Intelligence, Vol 129, No. 1-2*, pages 133–163. Elsevier Science, 2001.

[Nad90] Bernard A. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n-Queens. In *IEEE Expert: Intelligent Systems and Their Applications*, volume 5(3), pages 16–23. IEEE Educational Activities Department, 1990.

[Oll86] Kathleen Ollerenshaw. On most perfect or complete 8 x 8 pandiagonal magic squares. In *Royal Society of London*, volume 407, pages 259–281. 1986.

[OR97] John O'Connor and Edmund Robertson. Augustin Louis Cauchy, 1997. http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Cauchy.html.

[PB04] Steven Prestwich and Christopher Beck. Using pseudosymmetry to reduce search effort. Technical report, TR-01-2004, 2004. Available from http://4c.ucc.ie/4cite/TR-01-2004.pdf.

[Pea03] Justin Pearson. Comma-free codes. In Barbara M. Smith, Ian P. Gent, and Warwick Harvey, editors, *Symmetry in Constraint Satisfaction Problems*, pages 161–167, 2003.

[Pre00] Steven Prestwich. An Informal Tutorial on Search Techniques in Constraint Programming. In *Workshop on Information Integration and Web-based Applications & Services*, pages 105–121, 2000.

[Pre01] Steven Prestwich. First-Solution Search with Symmetry Breaking and Implied Constraints. In *Modelling and Problem Formulation - CP2001*, 2001.

[Pre02] Steven Prestwich. Supersymmetric Modelling for Local Search. In Piere Flener and Justin Pearson, editors, *SymCon'02: Symmetry in Constraint Satisfaction Problems*, pages 21–28, 2002.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. In *Computational Intelligence 9(3)*, pages 268–299. 1993.

[PS98] Les Proll and Barbara Smith. Integer linear programming and constraint programming approaches to a template design problem. In *INFORMS Journal on Computing*, volume 10, pages 265–275. 1998.

[PS03] Karen Petrie and Barbara M. Smith. Symmetry breaking in graceful graphs. Technical Report APES-56a-2003, APES, 2003.

[Pug93] Jean-François Puget. On the satisfiability of symmetrical constrained satisfaction problems. In J. Komorowski and Z. W. Ras, editors, *Methodologies for Intelligent Systems: Proc. of the 7th International Symposium ISMIS-93*, pages 350–361. Springer-Verlag, 1993.

[Pug02] Jean-François Puget. Symmetry breaking revisited. In P. van Hentenryck, editor, *Principles and Practice of Constraint Programming*, pages 446–461. Springer, 2002.

[Pug03] Jean-François Puget. Symmetry breaking using stabilizers. In F. Rossi, editor, *Principles and Practice of Constraint Programming*, pages 585–599. Springer, 2003.

[RDGKL04] Colva M. Roney-Dougal, Ian P. Gent, Tom Kelsey, and Steve Linton. Tractable symmetry breaking using restricted search trees. Technical Report 2004/3, Centre for Interdisciplinary Research in Computational Algebra, 2004.

[Reg94] Jean-Charles Regin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-94: Twelfth National conference on Artificial Intelligence*, pages 362–367, Seattle, WA, 1994.

[Reg96] Jean-Charles Regin. Generalized arc consistency for global cardinality constraints. In *American Association for Artificial Intelligence*, pages 209–215, 1996.

[Roy98] Gordon F. Royle. An orderly algorithm and some applications. In *Discrete Mathematics*, volume 185, pages 105–115. Elsevier Science, 1998.

[RPD90] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In Luigia Aiello, editor, *European Conference on Artificial Intelligence*, pages 550–556. Pitman, 1990.

[Sel03] Evgeny Selensky. A Reformulation of the Bridge Building Problem as Vehicle Routing. In Alan Frisch, editor, *Modelling and Reformulating Constraint Satisfaction Problems*, pages 132–142, 2003.

[Ser03] Akos Seress. *Permutation Group Algorithms*. Cambridge University Press, 2003.

[SG98] Barbara M. Smith and Stuart Grant. Trying Harder to Fail First. In H. Prade, editor, *European Conference on Artificial Intelligence*, pages 249–253, 1998.

[SLM92] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *AAAI-92: Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[Smi97] Barbara M. Smith. Succeed-first or fail-first: A case study in variable and value ordering heuristics. In *Practical Applications of Constraint Technology*, pages 321–330, 1997.

[Smi01] Barbara M. Smith. Reducing symmetry in a combinatorial design problem. Technical Report Research Report 2001.01, University of Leeds, January 2001.

[Soi99] Leonard Soicher. On the structure and classifications of somas: Generalizations of mutually orthogonal latin squares. In *Proceedings of The Electronic journal of Combinatorics*, page R32, Queen Mary and Westfield College, London, UK, 1999.

[Ste01] Kostas Stergiou. *Representation and Reasoning with Non-Binary Constraints*. PhD thesis, University of Strathclyde, 2001.

[SW99] Kostas Stergiou and Toby Walsh. Encodings of non-binary constraint satisfaction problems. In *American Association for Artificial Intelligence*, pages 163–168, 1999.

[vD82] Walther von Dyck. Gruppentheoretische studien. In *Math. Annalen*, volume 20, pages 1–44. 1882.

[vD03] Marc van Dongen. Lightweight arc-consistency algorithms. Technical report, TR-01-2003, 2003. Available from http://4c.ucc.ie/4cite/TR-01-2003.pdf.

[vEJMT99] C. A. J. van Eijk, E. T. A. F. Jacobs, B. Mesman, and A. H. Timmer. Identification and Exploitation of Symmetries in DSP Algorithms. In *Design, Automation and Test in Europe*, pages 602–608. IEEE Computer Society, 1999.

[Wal93] Richard Wallace. Why AC-3 is Almost Always Better than AC-4 for Establishing Arc Consistency in CSPs. In Ruzena Bajcsy, editor, *International Joint Conference on Artificial Intelligence*, pages 239–245. Morgan Kaufmann, 1993.

[WNS97] Mark Wallace, Stefano Novello, and Joachim Schimpf. ECL^{i}PS^e: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, 1997.

[ZY01] Yuanlin Zhang and Roland Yap. Making AC-3 an Optimal Algorithm. In *International Joint Conference of Artificial Intelligence*, pages 316–321, Seattle, WA, 2001.

Appendix A

Glossary

AC - Arc Consistent (or Arc Consistency depending on the context)

CSP - Constraint Satisfaction Problem

GHK-SBDS - A modified version of the Symmetry Breaking During Search technique by Gent, Harvery and Kelsey [GHK02]

GHKL-SBDD - A new implementation of Symmetry Breaking via Dominance Detection technique Gent, Harvey, Kelsey and Linton [GHKL03]

PSB - Partial Symmetry Breaking

SBDD - Symmetry Breaking via Dominance Detection [FSS01]

SBDS - Symmetry Breaking During Search [GS00]

Appendix B

NuSBDS - User Manual

B.1 What is NuSBDS

NuSBDS is an archive that can be used with ILOG Solver to perform symmetry breaking during search. It allows the constraint programmer to state the symmetries acting on the variables and/or assignments in their CSP and ILOG Solver will generate symmetrically equivalent solutions based on the SBDS technique developed by Gent and Smith. More specifically, NuSBDS also uses some techniques developed by Gent, Harvey and Kelsey using Computational Group Theory (CGT). Future versions of NuSBDS will contain other methods from other research into symmetry in constraint programming.

Since this is a beta version there are still a few bugs that have yet to be fixed. There are probably quite a few more that haven't been spotted yet. If you find any, it would be greatly appreciated if you could send details to `iain@dcs.st-and.ac.uk`.

That said, if your CSPs are highly symmetric then NuSBDS should yield large savings in run-time. The remainder of this user manual talks you through installing NuSBDS to using the NuSBDS symmetry macros to automate describing symmetries.

B.2 Installing NuSBDS

B.2.1 Compiling NuSBDS

If you already have a pre-compiled archive for your system then you can go directly to Section B.2.2. If not then this section will tell you how to produce your own archive.

Firstly, create a directory with the following files: `Group.cpp`, `Group.h`, `Symmetry.cpp`, `Symmetry.h`, `SymCon.cpp`, `SymCon.h`, `Search.cpp` and `Search.h`. Create a makefile based on your default ILOG Solver makefile and add this file to your directory. Add the following to your makefile:

```
Group.o:Group.cpp
    $(CCC) -c $(CFLAGS)  -o Group.o Group.cpp

Symmetry.o:Symmetry.cpp
    $(CCC) -c $(CFLAGS)  -o Symmetry.o Symmetry.cpp

SymCon.o:SymCon.cpp
    $(CCC) -c $(CFLAGS)  -o SymCon.o SymCon.cpp

Search.o:Search.cpp
    $(CCC) -c $(CFLAGS)  -o Search.o Search.cpp
```

The variables `CCC` and `CFLAGS` should already be defined in your makefile. Consult the example ILOG Solver makefile for more details.

Go to the relevant directory that contains the NuSBDS source code and type the following at the command prompt:

```
[user@host nu_sbds_dir]$ make Group.o
[user@host nu_sbds_dir]$ make Symmetry.o
[user@host nu_sbds_dir]$ make SymCon.o
[user@host nu_sbds_dir]$ make Search.o
```

You can now create your NuSBDS archive. Depending on your version of ar you may need to use the `ranlib` command as well.

```
[user@host nu_sbds_dir]$ ar r sbds.a Group.o Symmetry.o SymCon.o Search.o
[user@host nu_sbds_dir]$ ranlib sbds.a
```

You should now have a working archive.

B.2.2 Installing Archive

Place your NuSBDS archive in the same directory as your `*.o` and executable files and place the NuSBDS `.h` files in the same directory as your CSP source files. Next edit your makefile. If you have a CSP encoding in nqueen.cpp that you wish to use NuSBDS with, change your makefile entry for nqueen from this:

```
nqueen: nqueen.o
    $(CCC) $(CFLAGS)  nqueen.o -o nqueen  $(LDFLAGS)
nqueen.o: nqueen.cpp
    $(CCC) -c $(CFLAGS) -o nqueen.o nqueen.cpp
```

To this:

```
nqueen: nqueen.o sbds.a
    $(CCC) $(CFLAGS)  nqueen.o sbds.a -o nqueen  $(LDFLAGS)
nqueen.o: nqueen.cpp
    $(CCC) -c $(CFLAGS) -o nqueen.o nqueen.cpp
```

You are now ready to begin altering your CSP to make use of NuSBDS.

B.3 Using NuSBDS

Any ILOG Solver program that wished to use NuSBDS should contain the following line:

```
#include "Search.h"
```

You will also need an ILOCPGOALWRAPPER2 to call the functions in Search.h e.g.

```
ILOCPGOALWRAPPER2
    (Nu_SBDSGenerate, solver, IloIntVarArray, vars, Symmetry, sym){
    return IlcGenerateAlgebra(solver.getIntVarArray(vars), sym);
}
```

As well as the function IlcGenerateAlgebra(IlcIntVarArray, Symmetry) there are also the functions IlcGenerateAlgebra(IlcIntVarArray, Symmetry, IlcChooseIntIndex) and IlcGenerateAlgebra(IlcIntVarArray, Symmetry, IlcChooseIntIndex, IlcIntSelect which allow the user to supply their own variable and value ordering heuristics. See Search.h for more details.

A Symmetry object needs to be created like this:

```
IloEnv env;
Symmetry sym(env);
```

The NuSBDS library is finally used like so:

```
IloModel mdl(env);
IloSolver solver(mdl);
IloIntArray type(env, 1, SQUARE);
IloGoal goal = Nu_SBDSGenerate
    (Nu_SBDSGenerate(env, x, sym.setup(x, solver, ASSIGN, type)));
solver.startNewSearch(goal);
```

This by no means explains how to use NuSBDS but it highlights how easy it is to use NuSBDS. With just a few extra lines of code it is possible to perform symmetry breaking in CSPs.

B.3.1 NuSBDS in Practice

The following is an encoding of the n-queens problem in ILOG Solver. This CSP has been modified to make use of NuSBDS in order to break the 8 symmetries of this problem i.e. the symmetries of a square.

```
#include <ilsolver/ilosolverint.h>
#include "Search.h"

ILOSTLBEGIN

IlcChooseIndex2(IlcChooseMinSizeMin,
    var.getSize(),
    var.getMin(),
    IlcIntVar)

ILOCPGOALWRAPPER1(MyGenerate, solver, IloIntVarArray, vars) {
    return IlcGenerate(solver.getIntVarArray(vars), IlcChooseMinSizeMin);
}

ILOCPGOALWRAPPER2
    (Nu_SBDSGenerate, solver, IloIntVarArray, vars, Symmetry, sym){
    return IlcGenerateAlgebra(solver.getIntVarArray(vars), sym,
    IlcChooseMinSizeMin);
}

int main(int argc, char** argv) {
    IloEnv env;

    try {
        IloModel mdl(env);

        IloInt nqueen = (argc > 1) ? atoi(argv[1]) : 8;
        IloInt symbreaking = (argc > 2) ? atoi(argv[2]) : 1;
        IloIntVarArray x (env, nqueen, 0, nqueen-1);
```

```
IloIntVarArray x1(env, nqueen, -2*nqueen, 2*nqueen);
IloIntVarArray x2(env, nqueen, -2*nqueen, 2*nqueen);


IloInt i;
for (i = 0; i < nqueen; i++) {
    mdl.add(x1[i] == x[i]+i);
    mdl.add(x2[i] ==  x[i]-i);
}

mdl.add(IloAllDiff(env, x));
mdl.add(IloAllDiff(env, x1));
mdl.add(IloAllDiff(env, x2));

IloSolver solver(mdl);
Symmetry sym(env);

IloGoal goal;
if(symbreaking){
    IloIntArray type(env, 1, SQUARE);
    goal = Nu_SBDSGenerate
        (env, x, sym.setup(x, solver, ASSIGN, type));
} else{
    goal = MyGenerate(env, x);
}

solver.startNewSearch(goal);
IloInt numOfSolutions = 0;

while (solver.next()) {
    numOfSolutions++;
}

solver.out() << numOfSolutions << " solutions" << endl;
```

```
        solver.endSearch();
        solver.printInformation();
    }
    catch (IloException& ex) {
        cerr << "Error: " << ex << endl;
    }
    env.end();
    return 0;
}
```

B.4 Using the NuSBDS macros

The n-queens example in the previous section used one the NuSBDS macros to describe the symmetries of the problem. This was done by the following two lines:

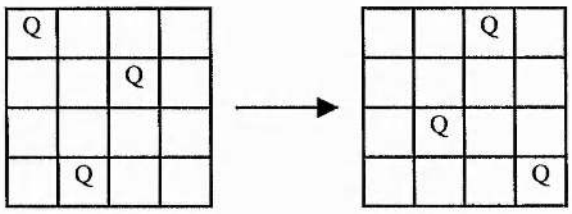
```
IloIntArray type(env, 1, SQUARE);
goal = Nu_SBDSGenerate(env, x, sym.setup(x, ASSIGN, type));
```

The first specifies that this CSP has "1" type of symmetry and that it is the symmetry of a "SQUARE". The second line specifies that the symmetry acts on the "ASSIGNMENTS" rather than the variables. Both `SQUARE`, `ASSIGN` and all further capitalized variables are defined in `Symmetry.h`

There are three advantages of using the NuSBDS macros. Firstly, even though NuSBDS used CGT techniques to break symmetry, the constraint programmer does not need to know any group theory in order to express symmetries. Secondly, changing the size of the problem does not change the nature of the symmetries but it does change the group acting on the problem. Using the macros means the group is calculated with respect to the size of the problem every time so the whole process of describing the group is automated for any size of problem. Finally, combinations of symmetries can be described by using more than one macro e.g. the most perfect magic squares problem has the symmetries of a square as well as being able to cycle the rows and cycle the columns. This can be represented by creating the following array: `IloIntArray type(env, 3, SQUARE, CYCLE_ROW, CYCLE_COL)`.

B.4.1 How to spot the symmetries

In order to use NuSBDS, the symmetries of the relevant CSP need to be recognized. This information needs to be related to the NuSBDS syntax. Here is an example with the n-queens problem.



In this example we can see that the partial assignment $Q_1 = 1$ & $Q_2 = 3$ & $Q_4 = 2$ is translated to the partial assignment $Q_1 = 3$ & $Q_3 = 2$ & $Q_4 = 4$ by rotating the board 180°. In this example we can see that the symmetries act on the "assignments" i.e. the symmetries alter not just the variable but also their value. Therefore in this case, when we use the `Symmetry::setup(IloIntVarArray, IloInt, IloIntArray)` method, the second argument should be ASSIGN and not VAR. Now consider the BIBD problem.

0	0	0	1	1	
0	0	1	1	0	
1	0	1	0	1	

Here each square represents a boolean variable and not an assignment. Numbering the squares from left to right and top to bottom, if we swap column 1 and column 2, $var_1 \Leftrightarrow var_2$, $var_6 \Leftrightarrow var_7$, $var_{11} \Leftrightarrow var_{12}$. The values of each variable remains unchanged thus we say the symmetries act on the "variables".

B.4.2 Available NuSBDS macros

Here are the available NuSBDS macros for symmetries on assignments:

- SQUARE - e.g. n-queens
- SYMMETRIC_VAR - interchangeable variables
- SYMMETRIC_VAL - interchangeable values
- SQUARE_VAR - n^2 variables with the symmetry of an n by n square acting on them

Here are the available NuSBDS macros for symmetries on variables:

- SQUARE - n^2 variables with the symmetry of an n by n square acting on them
- CYCLE_ROW - n^2 variables make a square where the rows can be cycled
- CYCLE_COL - n^2 variables make a square where the columns can be cycled
- SYMMETRIC_ROW - n^2 variables make a square where the rows are interchangeable
- SYMMETRIC_COL - n^2 variables make a square where the columns are interchangeable

If you have variables that form a rectangle and not a square e.g. BIBD problems, then the following macros may also be relevant:

- SYMMETRIC_RECTANGLE_ROW - as SYMMETRIC_ROW but for rectangles

- SYMMETRIC_RECTANGLE_COL - as SYMMETRIC_COL but for rectangles

Where the symmetries act on variables which form squares e.g. n-queens, the dimensions of the specific square can be calculated from the size of the array of constrained integer variables. If the variables form a rectangle, NuSBDS needs to be told how many columns there are before the above two macros can be used. This is done by using the method `void Symmetry::setNumOfColumns(IloInt)`.

NuSBDS performs some checks to ensure that the symmetries you have described are valid for your array of constrained variables. However, it is advised that you validate your solutions to ensure that you have used the macros correctly.

If there are any types of symmetry not covered here, email `iain@dcs.st-and.ac.uk` with a description of the symmetry and a macro for it may appear in a future version.

Acknowledgements

Thanks to Steve Linton for help with Group Theory algorithms and making sure my code was efficient. Thanks to Andrew Rowley for sharing his expert C++ knowledge. Thanks to Ian Gent for explaining the GHK-SBDS algorithm. Thanks to Tom Kelsey for helpful discussions and suggestions.

Appendix C

Problems

In this appendix we will describe some of the common CSPs mentioned in this thesis. Many of these problems will be solved in order to present empirical evidence of different symmetry breaking techniques. The description of the problems will include information about potential models and the symmetries they contain.

C.1 Alien Tiles

This problem is problem number 27 in CSPLib [GW99]. More about alien tiles can be found at http://www.alientiles.com/.

Example C.1 *The alien tiles board can be described with two parameters n and c, the size of the board and the number of colours respectively. An alien tiles board is an $n \times n$ grid of n^2 coloured squares. By clicking on any square on the board, the colour of the square is changed $+1 \mod c$. As well as this, the colour of every square in the same row and column is also altered $+1 \mod c$. Given an initial state and a goal state, the problem is to find the required number of clicks on each square which can be anything between 0 and $c-1$ (since $0 \equiv c$, $1 \equiv c+1$ etc). The more challenging problem for constraint programming (which we will consider) is finding the most complicated goal state (in terms of the number of clicks needed) for some initial state and then reaching that goal state in as few clicks as possible and verifying optimality.*

The model is quite simply n^2 variables with a domain from 0 to $c-1$. The expression of the constraints for this problem is quite complicated and required the use of GAP [GLS00].

Given a solution we can freely permute the rows and columns and flip the board around a diagonal. For a board with n^2 variables, the group acting on the board contains $2n!^2$ symmetries.

C.2 Balanced Incomplete Block Design

This problem is problem number 28 in CSPLib [GW99]. Balanced incomplete block designs have uses in, among other things, cryptography and experiment design [Far02].

Example C.2 *A balanced incomplete block design (BIBD) is a $v \times b$ binary matrix with exactly r ones per row, k ones per column, and with a scalar product of λ between any pair of distinct rows. A BIBD is therefore specified by its parameters (v, b, r, k, λ).*

Note that $vr = bk$ and $\lambda(v-1) = r(k-1)$. The most common model for the BIBD problem is a matrix model of binary variables. Such a model has a large amount of symmetries since both the rows and columns can be freely permuted. This results in $v! \times b!$ symmetries.

C.3 Dodecahedron Colouring

A dodecahedron colouring problem is a specific instance of a graph colouring problem. In this case the graph has the same shape as a dodecahedron where the vertices of the dodecahedron are the nodes of the graph, and the edges of the dodecahedron are the edges of the graph.

Example C.3 *A graph colouring problem consists of an undirected graph G, and a set of colours k. Each node in G must be given one of the k colours such that no two nodes in G that share an adjoining edge have the same colour.*

A standard model for this problem has each node in the graph represented as a variable with domain 1 to k. The constraints are simply that adjoining nodes cannot be equal.

Given k colours, there are $k!$ symmetries. These come from the permutations of the colours of the nodes. If the graph in question has any automorphisms then they can also be combined with the $k!$ value symmetries. The dodecahedron has 60 automorphisms. Thus, for a 3-colouring of a dodecahedron, there are $60 \times 3! = 360$ symmetries.

C.4 Fractions Puzzle

This problem is a small equation containing 9 unknown terms.

Example C.4

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1$$

Find values for each variable such that the equation is satisfied, and each letter has a different value from 1 to 9.

We can model this problem quite simply by selecting each letter as a variable, whose domain is the numbers from 1 to 9. If we consider the three seperate fractions of this problems, we can see that the commutative operator $+$ is acting on them. Therefore a solution to this problem can be permuted by any of the 3!, or 6, re-orderings of these fractions.

C.5 Golfers' Problem

This problem is problem number 10 in CSPLib [GW99]. It is based on a question posed on the sci.op-research newsgroup in May 1998 for a real golf tournament scheduling problem. Though the specific instance referred to contained 32 golfers, and 8 groups of 4, it can be generalised to other sizes.

Example C.5 *Given p players, and g groups of golfers (where $p \bmod g = 0$), schedule these groups of golfers over w weeks, such that no golfer plays in the same group as any other golfer twice.*

Note that every week, a golfer must play with $\frac{p}{g} - 1$ new golfers, and there are only $p - 1$ other golfers. Therefore $w \leq \frac{p-1}{\frac{p}{g}-1}$.

The golfers' problem is a very important example for general symmetry breaking methods as it has a great deal of symmetry resulting from complex interactions. We can freely permute the order of the groups within any week. The labelling of the players can be permuted freely. As well as this, we can re-order the weeks of any solution to yield another. All of these symmetries can be combined. This results in not just a highly symmetric problem, but also one whose symmetries are difficult to describe. Since there are many models to the golfers' problem [FSS01] [Smi01], each with their own number of symmetries, we will discuss the exact number of symmetries of specific models when relevant.

C.6 Most Perfect Magic Squares

A magic square is an $n \times n$ grid of unique numbers such that the sum of the rows, columns and diagonals all add to the same number. A most perfect magic square is a more complex structure with tighter constraints.

Example C.6 *A most perfect magic square is an $n \times n$ grid of numbers with different values from 1 to n^2 that satisfy the following constraints:*

1. *Each row and column sum to $(n^3 + n)/2$*
2. *Every 2×2 block of cells (including wrap-around) sum to $2T$ (where $T = n^2 + 1$).*
3. *Any pair of integers distant $\frac{1}{2}n$ along a diagonal (including wrap-around) sum to T.*

The model is a straightforward matrix of n^2 variables with a domain of 1 to n^2. This problem has $8n^2$ symmetries which are derived from the symmetries of a square combined with being able to cycle the rows and columns.

Though more complex symmetries can be found, the symmetries need a redundant model to be effectively realised [Oll86]. Breaking only a subset of all symmetries is still a valid way of reducing computation.

C.7 n-queens

This is a very common problem to describe as a CSP. Not only is this a simple problem to solve, it is also easy to explain and so is commonly used as an example problem to explain a simple concept. This problem also contains some symmetry and thus can be used as an example problem for symmetry breaking too.

Example C.7 *Given an $n \times n$ chess board, place n queens on it such that none can attack each other. In other words, no two queens can be place on the same row, column or diagonal.*

Various models are discussed in Chapter 2.1.1. Model 2.3 contains 8 symmetries. These are the symmetries of a square in 2 dimensional space i.e. given a solution, we can rotate the board by 90° (or another symmetry of a square) to yield another solution.